LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-538

μFX/DLX
A PEDAGOGIC COMPILER

[redacted]

Douglas Grundman
Raymie Stata
James O'Toole

[redacted]

March 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
3/30/92

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
μFX/DLX — A Pedagogic Compiler

5. FUNDING NUMBERS

6. AUTHOR(S)
Douglas Grundman, Raymie Stata, James O'Toole (Editor)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
MIT/LCS/TR-538

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA
1400 Wilson Blvd.
Arlington, VA 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N00014-89-J-1988

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report provides an overview of the μFX/DLX compiler. The source language for the compiler is μFX, a Lisp dialect that is a subset of FX-91. μFX is statically typed, and employs a type reconstruction algorithm to eliminate the need for type declarations. The compiler produces assembly code for the DLX, a simplified RISC architecture introduced by Patterson and Hennessy in their text, *Computer Architecture: A Quantitative Approach.*

μFX/DLX was written for the primary purpose of conducting experiments concerning basic features of programming language implementation. For example, it is used in the MIT graduate-level programming language course, where students are expected to read, understand, and modify the compiler, in order to investigate the effects of various optimizations.

The organization of the compiler and its intermediate forms are described via examples. The register usage, memory layouts, and calling conventions are explained. Some suggested experiments are presented, and the annotated implementation of μFX/DLX is provided. The report is not entirely self-contained, as it does not completely describe the details of the source and target languages.

14. SUBJECT TERMS
Programming Languages, Types, Effects, Inference, Polymorphism
Compilation, Optimization

15. NUMBER OF PAGES
165

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

μFX/DLX — A Pedagogic Compiler

Douglas Grundman
Raymie Stata
James O'Toole, Editor
Programming Systems Research Group
Laboratory for Computer Science

March 30, 1992

Summary

This report provides an overview of the μFX/DLX compiler. The source language for the compiler is μFX, a Lisp dialect that is a subset of FX-91. μFX is statically typed, and employs a type reconstruction algorithm to eliminate the need for type declarations. The compiler produces assembly code for the DLX, a simplified RISC architecture introduced by Patterson and Hennessy in their text, *Computer Architecture: A Quantitative Approach.*

μFX/DLX was written for the primary purpose of conducting experiments concerning basic features of programming language implementation. For example, it is used in the MIT graduate-level programming language course, where students are expected to read, understand, and modify the compiler, in order to investigate the effects of various optimizations.

The organization of the compiler and its intermediate forms are described via examples. The register usage, memory layouts, and calling conventions are explained. Some suggested experiments are presented, and the annotated implementation of μFX/DLX is provided. The report is not entirely self-contained, as it does not completely describe the details of the source and target languages.

Keywords: Programming Languages, Types, Effects, Inference, Polymorphism, Compilation, Optimization

This research was supported by the Defense Advanced Research Projects Agency of the Departement of Defense and was monitored by the Office of Naval Research under contract number N00014-89-J-1988

Preface

This report represents a combination of several sources of documentation concerning the pedagogical compilers which have been written for MIT EECS course 6.821. The first such compiler was written by Jonathan Rees. That compiler was used for several years between approximately 1987 and 1990. It was modified in minor ways by Franklyn Turbak, Mark Sheldon, and James O'Toole. Students made use of the compiler in the final problem sets of course 6.821.

During early 1991, Doug Grundman rewrote most of the compiler. Doug designed the intermediate code representations and wrote a new backend for the compiler which produced assembly code compatible with the DLX architecture [2]. During the summer of 1991, Raymie Stata joined the compiler project. Raymie and Doug rewrote portions of the compiler to improve performance, implemented tail-recursive call optimization, and corrected the generation of code which used the stack. Raymie also added support for more of the source language (μFX), and modified the garbage collector to avoid copying stack-allocated data.

At the time of this writing, the compiler and associated simulation software are being prepared for student use in the Fall 1992 semester of 6.821. Brian Reistad is improving the typechecking phase of the compiler. This document was compiled from three primary sources: Doug's general overview of the compiler phases, Raymie's description of register usage conventions, and older materials describing Rees's version of the compiler. These documents were merged; some additional text and figures were added.

All of the software and documentation described in this report is available in electronic form. We hope that this report will permit the reader to enjoy the 6.821 pedagogical compiler.

James O'Toole
Editor

Contents

1 Introduction

The MIT subject 6.821 is an introduction to programming language semantics and pragmatics for graduate students. In the laboratory portion of 6.821, students read and modify a simple compiler to gain hands-on experience with fundamental concepts in compilation.

We have built a new compiler for the μFX programming language for use in the class. The compiler, which generates code for Patterson and Hennessy's DLX architecture, was designed to be especially easy to understand and modify. The purpose of this report is to make the compiler more accessible by explaining its organization and providing an overview of various internal representations. Information for acquiring a copy of the μFX software package and a simulator for DLX is in Section 5. Any questions or bug reports concerning the software described in this report should be sent via electronic mail to `microfx@brokaw.lcs.mit.edu`.

The μFX/DLX compiler was designed with two goals in mind. First, there was the pedagogic goal: the compiler needed to be useful for teaching compilation techniques to students. The second goal was that the compiler had to afford easy experimentation by being easy to modify. That way, the compiler would be useful not only in the classroom, but in a research setting as well.

These two goals — readability and writability — overshadowed all others in the design of the compiler. μFX/DLX makes no pretense of being a production compiler. Code quality and space efficiency have been largely ignored in favor of intelligibility. For example, the compiler's code generator contains only a few special case code improvements, and these were added only to improve the readability of the emitted code.

These two primary goals determine that the compiler be as *simple* as possible, and as *modular* as possible. The former aspect says that the compiler contains no hidden intricacies to improve run-time performance or compilation speed, while the latter decomposes the compiler into several simple passes that interact only through well-defined and well-documented interfaces.

It follows that adding a new pass (such as an optimizer) is relatively easy, as is making modifications to any of the pre-existing passes. Experience has indeed shown this to be the case.

The remaining sections of this report discuss the features of the μFX language, introduce the reader to the DLX target machine, give an overview of the compiler, and tell where to obtain a copy of the software.

1.1 The μFX Language

μFX is a subset of the FX-91 programming language [1], and may be thought of as a cross between Scheme and ML. The μFX syntax is shown below.

μFX is lexically scoped, with all parameters passed by value. Like FX-91, μFX is strongly typed, incorporating an ML-style type reconstructor. The language has first-class procedures, tail-recursion, and garbage collection. Its primitive data types include *integers*, *characters*, *symbols*, *strings*, *references*, and *procedures*.

$I \in$ Identifier
$E \in$ Expression
$N \in$ Integer-Numeral
$B \in$ Boolean-literal = {`#t`, `#f`}
$S \in$ String-literal = character sequences delimited by double-quotes
$L \in$ Literal = Integer-Numeral $\cup$ Boolean-literal $\cup$ String-literal

E ::= L
| I
| `(lambda (`I^*`)` E_B`)`
| `(`E_0 E^*`)`
| `(let ((`I E`)*)` E_B`)`
| `(letrec ((`I E`)*)` E_B`)`
| `(ref` E`)`
| `(^` E`)`
| `(:=` E_1 E_2`)`
| `(if` E_1 E_2 E_3`)`
| `(and` E^*`)`
| `(or` E^*`)`
| `(begin` E^*`)`

The language supported by the compiler uses the S-expression style in order to simplify parsing and permit easy experimentation with new language features. A number of primitive procedures are supported by the compiler as part of the runtime library (see Appendix B).

1.2 The Target Machine

The DLX architecture was introduced by Patterson and Hennessy in their book, *Computer Architecture: A Quantitative Approach* [2]. DLX has a generic RISC instruction set very similar to that of the MIPS architecture. It has 32 32-bit general-purpose registers, 32 32-bit floating-point registers, and no condition codes. There is one data addressing mode: register indirect with (signed) 16-bit offset. All memory accesses must be aligned according to the size of the referenced datum, otherwise a trap occurs. DLX has a delay slot following each branch or trap instruction, but differs from the MIPS architecture in that it has load interlocks.

There is a publicly available simulator for DLX, called *dlxsim*. The μFX/DLX compiler emits code which runs directly on dlxsim, but also contains a built-in DLX emulator that suffices for running small test cases. The simulator for DLX may be obtained via anonymous ftp from `max.stanford.edu` in `pub/hennessy-patterson.software`.

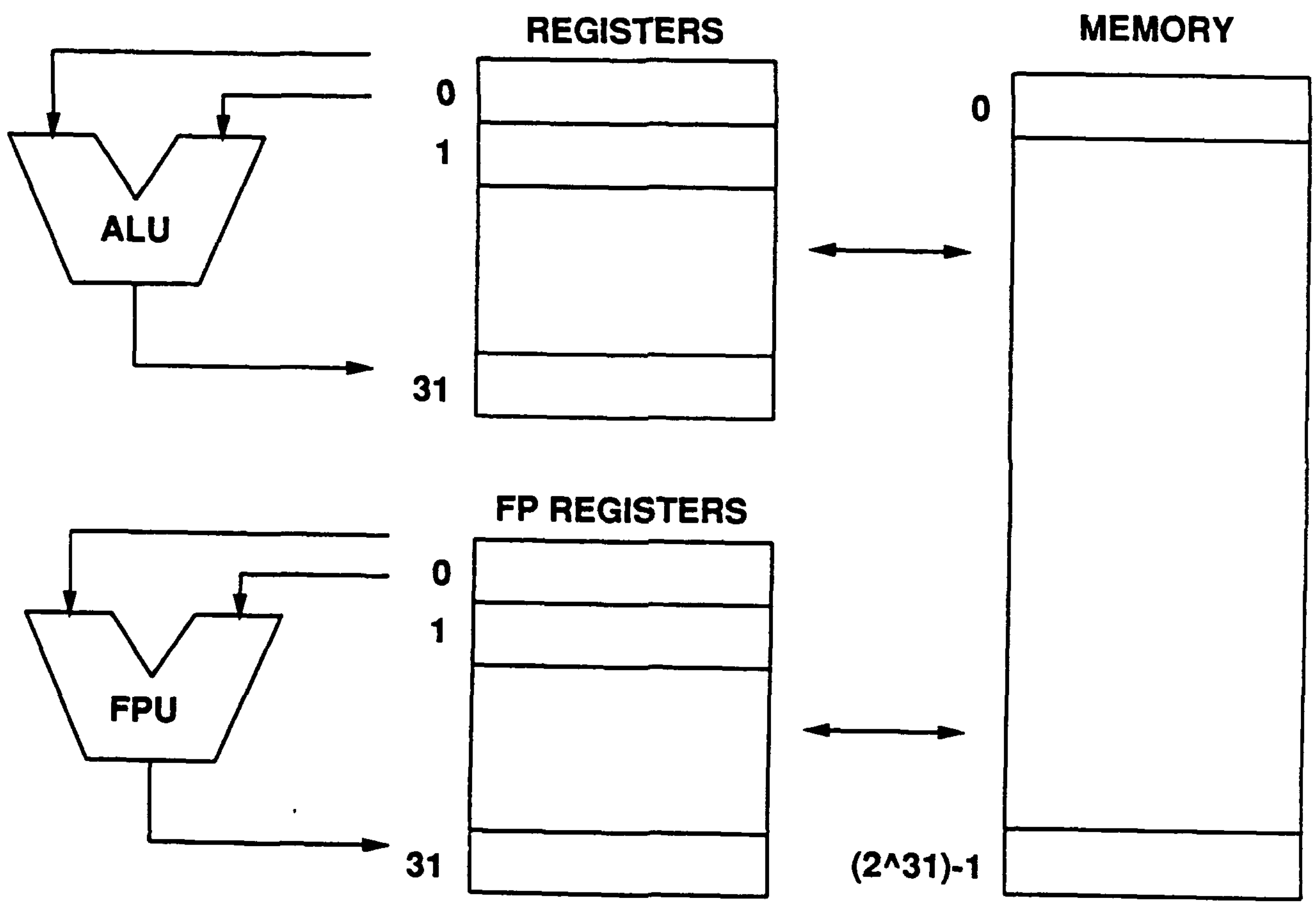

Figure 1: The DLX Processor

2 Compiler structure

μFX/DLX is organized into seven passes that make use of three intermediate forms: *exp*, *icode*, and *ocode*. For modularity, the compiler passes communicate only through these intermediate forms, which have been designed to be easy to understand and to work with so that students need not deal with unnecessary obstacles.

The seven compiler passes are:

parser: converts s-expressions to *exps*.
type reconstructor: annotates *exp* nodes with type information.
translator: converts *exps* to *icode*.
optimizer: *icode* to (improved) *icode*.
code generator: converts *icode* to *ocode*.
peephole optimizer: *ocode* to (improved) *ocode*.
output stage: *ocode* to assembly-code text.

The parser, type reconstructor, translator, and code generator have been mentioned above. The local optimizer currently only implements tail calls. Although a peephole optimizer is not currently implemented, the compiler provides for one so that delay slot filling can be done. The final output stage takes care of formatting the instructions into an ASCII text representation so that dlxsim will accept the output file. It also supplements the compiled program with a pre-written run-time support system (see Appendix D), thus functioning as a simple linker.

The software comes with a built-in interpreter which can execute programs at the ocode level. This interpreter, though slow, provides an easy way to test new compiler features. It is outfitted with a run-time environment similar to that supplied for running under dlxsim. Figure 2 shows the organization of the various stages of the compiler and its associated ocode simulator.

2.1 Using the System

The system is written in mini-FX, which is a slightly larger subset of FX-91 than is μFX. Several commands have been implemented that allow the user to display the output of any stage of the compiler. All of these procedures take a single argument which is a μFX expression represented as an S-expression. This set of commands includes:

- **`(test-parse sexpr)`**: prints a freshly-parsed representation of the μFX expression sexpr.
- **`(check sexpr)`**: type-checks sexpr.
- **`(show-type-check sexpr)`**: type-checks sexpr and displays the parse tree annotated with the reconstructed type information.
- **`(itest-compile sexpr)`**: compiles sexpr and prints (unoptimized) icode.
- **`(otest-compile sexpr)`**: compiles sexpr and prints optimized icode.

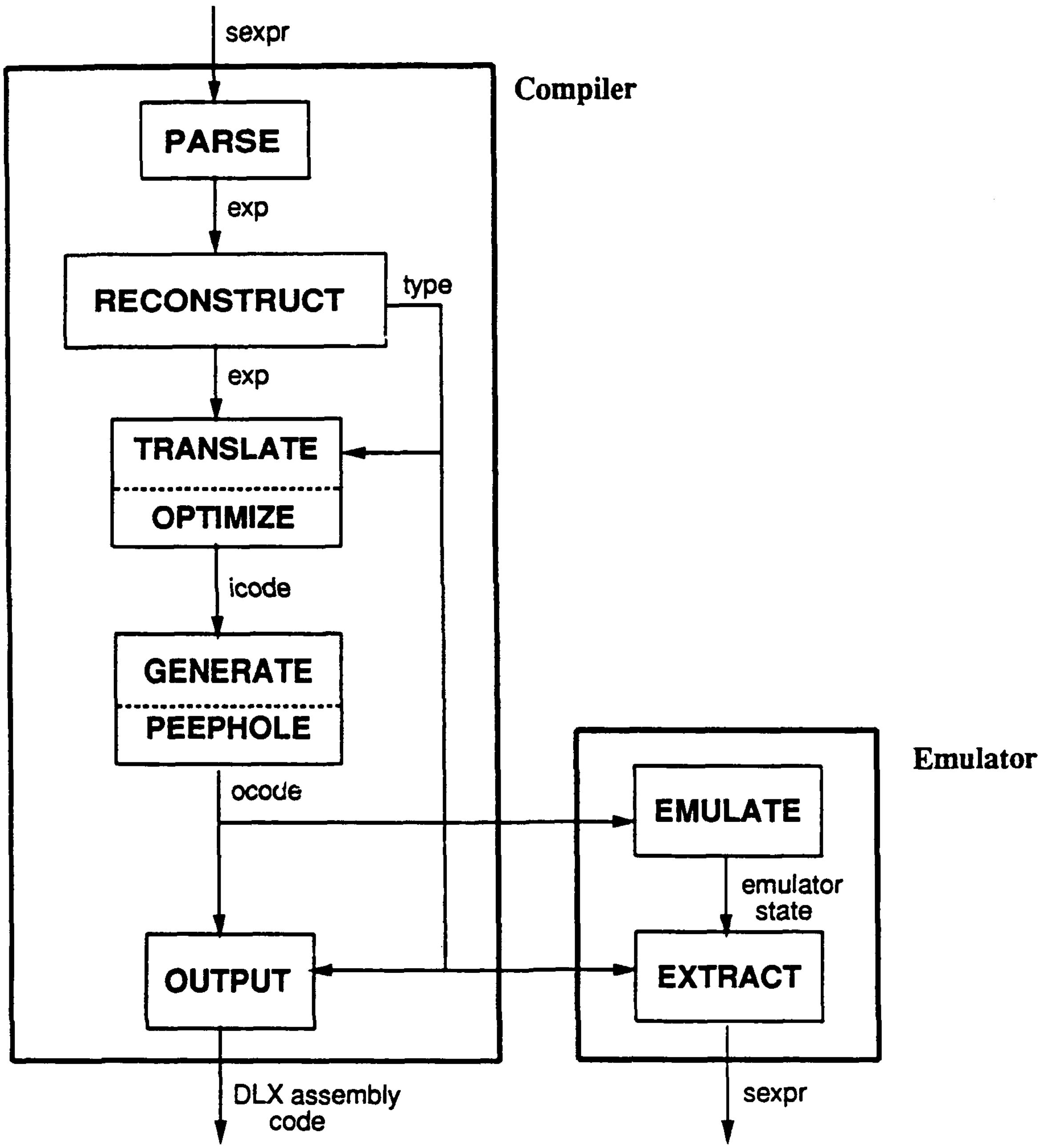

Figure 2: Organization of μFX/DLX

- **(test-compile sexpr)**: compiles sexpr and prints DLX assembly code. This is essentially ocode, printed in a more readable format.
- **(fx sexpr)**: compiles sexpr, emitting a runnable DLX program in the file fx.s in the current directory. This file is the output of `test-compile` with run-time support code added. It is suitable for loading directly into dlxsim.
- **(run sexpr)**: compiles and interprets sexpr, using the built-in ocode interpreter.

2.2 The "exp" Intermediate Form

The highest-level intermediate form — the "exp" form — is a parsed representation of the source language. It encodes primitives (booleans, characters, integers, strings, symbols and variables), abstractions, applications, conditionals, `let`, and `letrec`. The parser desugars all other language features into these forms.

```
> (test-parse '(lambda (x) x))
(ABSTRACTION->EXP TY (X) (VARIABLE->EXP TY X))
```

In the above example, the micro-FX program is quoted to make it a scheme s-expression. The parse tree which is returned indicates that this s-expression represents an abstraction. Inside the parse tree, TY stands for type information which will be filled in by the type-reconstructor.

Each exp tree node contains a field to hold the type of an expression. The type of each primitive constant is filled in at parse time; all others are filled in later by the type reconstructor.

```
> (test-parse '((lambda (x) x) 19))
(COMBINATION->EXP TY (ABSTRACTION->EXP TY (X) (VARIABLE->EXP TY X))
                     ((INT->EXP TY 19)))
```

In the above example, the closure is created and called with a list of arguments (19). The type reconstructor implements generic polymorphism [1].

```
> (check  '(lambda (x) x))
(-> (?X-1) ?X-1)
```

In this example, the type computed by the reconstruction phase is a function taking an argument of type ?X-1 and returning a value of type ?X-1. The notation ?X-1 represents an unbound type variable, and indicates that the type-checking problem is under-constrained. In other words, this function is polymorphic in the type of its one argument.

The `show-type-check` procedure shows the parse tree for an expression with full type information:

```
> (show-type-check '((lambda (x) x) 19))
(COMBINATION->EXP INT (ABSTRACTION->EXP INT (X) (VARIABLE->EXP INT X))
                  ((INT->EXP INT 19)))
```

In this example, the type of the argument and result of the lambda expression have been determined to be the same as the type of the integer constant 19.

2.3 The "icode" Intermediate Form

The middle-most intermediate form is termed "icode". Icode is designed to represent programs at the lowest possible level of abstraction without explicit register references. Type information present in the parse tree is also stored in the icode, although it is not normally printed when icode is displayed. The absence of explicit register references allows the issues of instruction selection and register allocation to be deferred to a subsequent pass. This goal was achieved with one exception: icode knows of the existence of an environment register so that the creation of closures can be represented.

```
> (itest-compile '(+ 1 2))

Type: int

Icode:

        START_1:
        (return (+ 1 2))
```

Here, START_1 is a label, and the "return" line is a tree printed (roughly) in preorder. START_1 always appears, and it is whe the execution of the compiled program begins.

```
> (itest-compile '((lambda (x) (+ x x)) 7))

Type: int

Icode:

    START_1:
        (return (call (alloc 2 ("LAMBDA_2" (r 2))) 7))
    LAMBDA_2:
        (body (1) (return (+ (var 0 1) (var 0 1))))
```

In this example, the lambda body has been separated from the call. (alloc 2 (...)) allocates 2 locations, initializing them with the values shown as arguments (the location LAMBDA_2 and register 2 (the previously-mentioned environment register)).

The instance of CALL shown here takes two "parameters": the first is the function (closure), and the second is the function's parameter. Finally, `(body (1) ...)` accepts one parameter, an expression, and runs it within a (newly-constructed) environment accepting 1 parameter. The icode form `(var 0 1)` is a reference to the variable `x`. For further explanation of environments and variable access, see section 3.2.1.

```
> (otest-compile '((lambda (x) (+ x x)) 7))

Type: int

Icode:
    START_1:
        (jump (alloc 2 ("LAMBDA_2" (r 2))) 7)
    LAMBDA_2:
        (body (1) (return (+ (var 0 1) (var 0 1))))
```

In this example, the `call`—`return` sequence has been replaced with a `jump`, thus implementing a tail-call. (JUMP has the same syntax as CALL.)

Any program expressed in icode takes the form of a list of labelled trees. The job of the compiler pass converting exps to icode is primarily to separate lambda bodies from the creation of corresponding closures. This explains why icode takes the form of a list of trees rather than a single tree: any expression containing a lambda sub-expression is split into two trees. One of these trees describes the creation of the closure, while the other describes the function's code.

2.4 The "ocode" Intermediate Form

The *lowest*-level intermediate form is termed "ocode". It has a 1-1 correspondence with DLX assembly code, and is converted to textual assembly code by the compiler's final pass. The code generator could have been designed to emit assembly-code directly, but the use of an intermediate form like ocode facilitates the construction of peep-hole optimizers, delay-slot fillers and the like.

```
> (test-compile '(+ 1 2))

Type: int

Object code:

Code:
    START_1:
        addi    ARG0, ZERO, 2
        addi    ARG0, ARG0, 4
        or      VAL, ARG0, ZERO
        lw      ATEMP, 11(FP)
        jr      ATEMP
        nop
```

This example shows the tiny program from above, fully compiled. The "1" and "2" show up as "2" and "4" in the object code because they are *tagged* values (see section 3.1.1). There is an assumption here that this code has been called with the standard calling convention. Both tagging and calling conventions will be explained more fully in section 3.2.

3 The Runtime Environment

The compiler comes with its own runtime library. Included in the library are a memory allocator (including a simple stop-and-copy garbage-collector), routines for saving and restoring registers to and from frames, and miscellaneous built-in primitives and I/O functions. The runtime library handles program invocation and termination, and prints the result of each computation and memory usage statistics after each run.

3.1 Memory organization

All heap and stack memory is organized into *blocks*. Each block is a sequence of four-byte *slots* whose address is a multiple of four. The main reason for this is that in DLX, as in most other modern RISC processors, all memory references to pointers or integers work only at four-byte boundaries. One particular slot in each block is called the "size" slot and the remaining slots are called the "data slots." The "size" slot is only accessible by the run-time system (which includes the memory allocator), while the data slots are accessible to the user's program. This use of explicit size information, though somewhat wasteful of memory, makes the system's garbage collector easier to read and understand.

The slots of a block are numbered starting from -1. Slot -1 is the size slot. Slot 0 is the first data slot; slot 1 is the second data slot; etc. Objects start at the lower address: the address of slot $i+1$ is always four bytes past the address of slot i.

The system supports two memory allocation primitives: _SALLOC, which allocates and zeroes a memory block from the stack, and _ALLOC, which allocates and zeroes a

memory block from the heap. If the stack memory allocator discovers that there is insufficient stack memory available with which to satisfy the current `_SALLOC` request, execution halts with an error. On the other hand, if the heap memory allocator discovers that there is insufficient heap memory available with which to satisfy the current `_ALLOC` request, it calls the run-time system's garbage collector in an attempt to discover memory that may be reused. If this attempt is successful, `_ALLOC` proceeds to allocate from the recycled memory, otherwise execution halts with an error.

3.1.1 Data Tagging

Any garbage collector needs to be able to distinguish pointers into the heap from miscellaneous integer values in the machine's registers. This is a run-time determination, and there are many ways that the compiler can help the garbage collector do this. The way used in the μFX/DLX compiler is that of *tagging* values at run-time. This relies on the convention that all data objects be aligned on even-byte boundaries (ours are aligned on four-byte-boundaries)

The *tag* of a (4-byte) data item is its lowest bit. Our tagging convention is that integers have a low bit of zero, while a pointer into the heap or stack has a low bit of one. This means the compiler has to adhere to two more conventions so that things all work. First, every integer n (and every atom representable in a single word) is represented by $2 \times n$. This makes the low bit of every integer a zero at the cost of decreasing the range of the integers we can represent and of complicating the code for multiplication and division (addition and subtraction work unchanged). Second, every pointer is represented by a word value that is one (byte) greater than the address of the object pointed-to. Since all objects are even-byte aligned, this makes all pointers (which did have a low bit of zero) have a low bit of one. The cost is that every memory reference through a pointer p must be adjusted at run-time to be a reference through $p - 1$.

Executable code in our model resides neither in the heap nor in the stack, so pointers into the code (such as return addresses) are never tagged. This agrees nicely with the semantics of DLX's jump-and-link instruction.

In the compiler code, the function `otag` (offset tag) provides a convenient way to name a slot; if p is a tagged pointer, `p + otag(-1)` is the machine address of the size slot, `p + otag(0)` is the machine address of the first data slot, etc.

3.1.2 Stack conventions

The stack is organized as a stack of four byte slots. Thus, the stack pointer is always moved in increments of four bytes. The stack register always contains the untagged address of the first free slot on the stack. The stack grows downward in the DLX address space.

Stack allocation of a block is done by subtracting the size of the object to be pushed in bytes, including the size slot from the stack pointer. A properly tagged pointer to the resulting block is the new stack pointer plus five. The runtime routine `_SALLOC` allocates and zeroes a block on the stack and accumulates statistics about stack space usage.

3.1.3 Garbage collection

Storage management is based upon a *stop-and-copy* garbage collector. In this scheme, The heap is divided into two equal-sized *semispaces*. At any time, one is considered "active" and the other "empty". Every heap allocation is made from the active semispace. When the memory allocator finds that the active semispace has filled up, it calls the garbage collector. The garbage collector swaps the active and empty semispaces, then scans all blocks pointed-to by any register and any blocks transitively accessible from any scanned block. It copies all of these accessible blocks into the (newly) active semispace. The garbage collector does not copy any memory block that is inaccessible, or any memory block on the stack (although any accessible blocks on the stack are scanned to discover new pointers into the heap).

The registers which start the copying process are called the *root set*. The root set includes all user data registers, the frame pointer, the environment pointer, and the temporary VAL register. These registers are discussed further in section 3.2. Of course, any register or block slot that does not contain a tagged pointer is not considered to be a pointer by the garbage collector.

Note that if a block in the stack is not reachable from the root set then it will not be scanned by the garbage collector. It is therefore safe to put data on the stack that is not in the standard memory block format, as long as this data does not include pointers which must be examined by the garbage collector. This is useful when writing the assembly code routines that interface to the operating system.

3.2 Calling convention

The caller is responsible for preserving the values of all registers except for VAL, ATEMP, and RETADR. Registers are saved in activation frames, which are linked dynamically and form the "dynamic chain." The head of the dynamic chain is pointed to by the frame pointer register FP (a tagged pointer).

Before each procedure call, an activation frame is allocated and linked into the dynamic chain. All registers (except VAL, ATEMP, and RETADR) are saved in this activation frame. Arguments to the procedure are evaluated and placed in registers ARG0, ARG1, etc.

By convention, at the time of a procedure call, register ARG0 holds a pointer to the procedure being called and ARGn holds the nth formal. When the procedure is called, the return address is stored into slot 2 of the activation record. The layout of the activation record is indicated in Figure 3. The live registers are restored from the current activation frame and computation continues.

At the end of a procedure body, the callee places the result in the VAL register and jumps to the return address stored in the activation frame. When the callee returns, the caller restores the stack pointer and frame pointer registers. Figure 4 shows the register usage of the μFX/DLX runtime system.

3.2.1 Representation of Environments

μFX/DLX environments are represented in memory by chains of blocks (called *ribs*). Indices 1 through n of each rib contain n values of a lexical environment's variables,

Offset	Contents	Comment
-1	frame size	Caller saves these registers. Put
0	FP (dynamic chain)	into frame when it is first alloc'd
1	SP (before frame was pushed!)	before a procedure invocation.
2	return address	
3	ENV	Caller saves these registers by invoking
4	r6	_SAVE just before arguments to
5	r7	callee are evaluated. Restored after
...	...	called procedure returns.
27	r29	

Figure 3: The Frame Layout

Mnemonic		Machine register number and Use
ZERO	0	Always zero — hardware convention
VAL	1	Val ret'd by proc, scratch, no one saves
ENV	2	Pointer to head of static chain, caller saves
FP	3	Pointer to head of dynamic chain, caller saves
SP	4	Stack pointer, caller saves
HP	5	Heap pointer, only used by ALLOC and GC
ARG0	6	Compiler temp, used to pass closure, caller saves
ARG1	7	Compiler temp, used to pass 1st argument, caller saves
ARG2	7	Compiler temp, used to pass 2nd argument, caller saves
ARG3	8	...ditto for register up to and incl. r29...
ATEMP	30	Scratch, pass args to system routines, no one saves
RETADR	31	Used as scratch, no one saves

Figure 4: DLX Register Usage

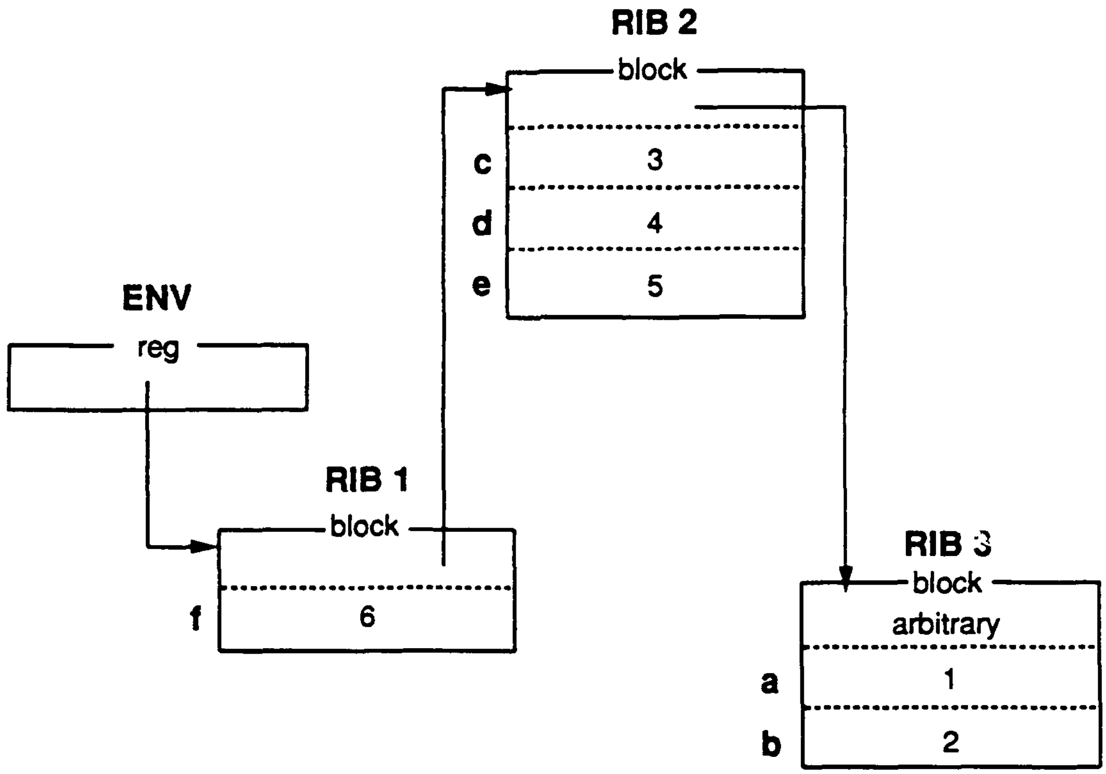

Figure 5: Example of Environment Representation

and index 0 contains a pointer to the parent environment. For example, consider a set of three nested let expressions:

```
(let ((a 1)
      (b 2))
  (let ((c 3)
        (d 4)
        (e 5))
    (let ((f 6))
      I_body)))
```

When I_{body} is evaluated, the environment structure is represented by the chain of blocks shown in Figure 5. The variable names shown in Figure 5 are not explicitly represented in the environment structure, only their values are. The names are shown in the figure for documentation purposes only.

Variables are accessed by traversing ribs "backward" until the proper rib is found, then by indexing "over" to set or retrieve the proper value. Thus, each accessible variable may be referenced from a code location by knowing only its "back" and "over" numbers. The compiler automatically maintains such numbers and does this for the user. In icode, a variable reference is encoded by the low-level primitive `(var back over)`. In figure 5, `f` is `(var 0 1)`, `c` is `(var 1 1)`, `d` is `(var 1 2)`, `e` is `(var 1 3)`, `a` is `(var 2 1)`, and `b` is `(var 2 2)`. In compiled code, `(var x y)`

translates to **x** indirect references through the environment pointer followed by an indexed access to the y-th component of that block.

4 Some Suggested Experiments

We have come up with a set of possible projects that illustrate various problems in compilation, some of which are in use here at MIT. Our list includes:

- Implementing stack allocation of frames and of environments when analysis shows this to be possible.
- The (re-)implementation of any one of the existing compiler phases (type reconstruction and code generation are good candidates).
- Replacing static links with displays.
- Allocating closures statically, or eliminating their construction altogether when analysis shows this to be possible.
- Modify activation record allocation to save only the registers which are live at the time of the call.
- Modify heap allocation to be performed inline instead of always calling _ALLOC.
- Adding new language features.

Further ideas for easy yet illustrative experiments for students would be valued.

5 Obtaining the Distribution in Electronic Form

Our system may be obtained via anonymous ftp from host `brokaw.lcs.mit.edu` in the directory `pub/microfx`. The package includes documentation, the compiler, run-time library, ocode interpreter, and a compiler test suite. Mini-FX is available via anonymous ftp from the same host.

Appendices C and D contains the source code of the compiler and the run-time library. This material is indexed by procedures, types, and variable names at the end of this report. This appendix represents a snapshot as of February 12, 1992 of the 25 Mini-FX source files which are available via FTP.

The simulator for DLX may be obtained via anonymous ftp from the host `max.stanford.edu`, in `pub/hennessy-patterson.software`, and is included in the `pub/microfx` distribution.

A The DLX Instruction Set

This appendix contains a brief description of the DLX instruction set, as used in μFX/DLX. This description is based on the assembler and interpreter supplied with dlxsim.

A.1 Registers and Miscellany

32 32-bit integer registers (r0..r31); r0 always reads as 0. 32 32-bit floating-pt registers (f0..f31) (doubles use even-odd pairs). There are no condition code bits. DLX has load interlocks. All loads and stores trap on unaligned access.

A.2 Integer Instructions

Operator	Operands	Comment
add	d,s1,s2	[traps on overflow]
addi	d,s,i16	[traps on overflow] (sign-extended i16 DATA)
addu	d,s1,s2	
addui	d,s,i16	
and	d,s1,s2	
andi	d,s,i16	
beqz	r,label	branch if reg is zero.
bnez	r,label	branch if reg is non-zero.
j	label	
jal	label	leaves address of instr after delayed instr in r31.
jr	r	s&3=0 else trap.
jalr	r	s&3=0 else trap. rct address like jsr.
lb	d,i16(s)	(sign-extended i16 OFFSET) sxt data byte to word
lbu	d,i16(s)	(sign-extended i16 OFFSET) zeroes high bits
lh	d,i16(s)	(sign-extended i16 OFFSET) sxt data halfword to word
lhi	d,i16	load immediate halfword << 16. zeroes low 16 bits
lhu	d,i16(s)	(sign-extended i16 OFFSET) zeroes high bits
lw	d,i16(s)	(sign-extended i16 OFFSET)
nop		

Operator	Operands	Comment
or	d,s1,s2	
ori	d,s,i16	
sb	i16(s),d	(sign-extended i16 OFFSET)
seq	d,s1,s2	
seqi	d,s,i16	(sign-extended i16 DATA)
sequ	d,s1,s2	same as seq
sequi	d,s,i16	different from seqi (no sign-ext of imm data here)
sge	d,s1,s2	
sgei	d,s,i16	(sign-extended i16 DATA)
sgeu	d,s1,s2	
sgeui	d,s,i16	
sgt	d,s1,s2	
sgti	d,s,i16	(sign-extended i16 DATA)
sgtu	d,s1,s2	
sgtui	d,s,i16	
sh	i16(s),d	(sign-extended i16 OFFSET)
sle	d,s1,s2	
slei	d,s,i16	(sign-extended i16 DATA)
sleu	d,s1,s2	
sleui	d,s,i16	
sll	d,s1,s2	
slli	d,s,i16	
slt	d,s1,s2	
slti	d,s,i16	(sign-extended i16 DATA)
sltu	d,s1,s2	
sltui	d,s,i16	
sne	d,s1,s2	

Operator	Operands	Comment
snei	d,s,i16	(sign-extended i16 DATA)
sneu	d,s1,s2	same as sne
sneui	d,s,i16	different from snei (no sign-ext of imm data here)
sra	d,s1,s2	
srai	d,s,i16	
srl	d,s1,s2	
srli	d,s,i16	
sub	d,s1,s2	(traps on overflow)
subi	d,s,i16	(traps on overflow) (sign-extended i16 DATA)
subu	d,s1,s2	
subui	d,s,i16	
sw	i16(s),d	(sign-extended i16 OFFSET)
trap	i27	
xor	d,s1,s2	
xori	d,s,i16	

A.3 Pseudo-Integer Instructions

Operator	Operands	Comment
div	fd,fs1,fs2	traps on div-by-zero
divu	fd,fs1,fs2	traps on div-by-zero
movfp2i	d,f	
movi2fp	f,r	
mult	fd,fs1,fs2	traps on overflow
multu	fd,fs1,fs2	traps on overflow

A.4 Floating Point Instructions

Operator	Operands	Comment
addd	Fd,Fs1,Fs2	
addf	fd,fs1,fs2	
bfpf	label	
bfpt	label	
cvtd2f	f,F	
cvtd2i	f,F	
cvtf2d	F,f	
cvtf2i	f1,f2	
cvti2d	F,f	
cvti2f	f1,f2	
divd	Fd,Fs1,Fs2	
divf	fd,fs1,fs2	
eqd	F1,F2	
eqf	f1,f2	
ged	F1,F2	
gef	f1,f2	
gtd	F1,F2	
gtf	f1,f2	
ld	F,i16(s)	(sign-extended i16 OFFSET)
led	F1,F2	
lef	f1,f2	
lf	f,i16(s)	(sign-extended i16 OFFSET)
ltd	F1,F2	
ltf	f1,f2	
movd	F1,F2	
movf	f1,f2	
multd	Fd,Fs1,Fs2	
multf	fd,fs1,fs2	
ned	F1,F2	
nef	f1,f2	
sd	i16(s),F	(sign-extended i16 OFFSET)
sf	i16(s),f	(sign-extended i16 OFFSET)
subd	Fd,Fs1,Fs2	
subf	fd,fs1,fs2	

B μFX/DLX Run-Time Library

This appendix contains a listing of the standard library routines supported by the runtime system of μFX/DLX. This is a subset of the FX-91 standard library.

Name	Type	Comment
symbol	(-> (name) symbol)	builtin constructor
and	(-> (bool bool) bool)	builtin operator
or	(-> (bool bool) bool)	builtin operator
backspace	char	constant
newline	char	constant
page	char	constant
space	char	constant
tab	char	constant
equiv?	(-> (bool bool) bool)	
and?	(-> (bool bool) bool)	
or?	(-> (bool bool) bool)	
not?	(-> (bool) bool)	
not	(-> (bool) bool)	
char=?	(-> (char char) bool)	
char<?	(-> (char char) bool)	
char>?	(-> (char char) bool)	
char<=?	(-> (char char) bool)	
char>=?	(-> (char char) bool)	
char-ci=?	(-> (char char) bool)	
char-ci<?	(-> (char char) bool)	
char-ci>?	(-> (char char) bool)	
char-ci<=?	(-> (char char) bool)	
char-ci>=?	(-> (char char) bool)	
char-alphabetic?	(-> (char) bool)	
char-numeric?	(-> (char) bool)	
char-whitespace?	(-> (char) bool)	
char-lower-case?	(-> (char) bool)	
char-upper-case?	(-> (char) bool)	
char-upcase	(-> (char) char)	
char-downcase	(-> (char) char)	
char->int	(-> (char) int)	
int->char	(-> (int) char)	

Name	Type	Comment
=	(-> (int int) bool)	builtin predicate
<	(-> (int int) bool)	builtin predicate
>	(-> (int int) bool)	builtin predicate
<=	(-> (int int) bool)	builtin predicate
>=	(-> (int int) bool)	builtin predicate
+	(-> (int int) int)	builtin operator
-	(-> (int int) int)	builtin operator
	(-> (int int) int)	builtin operator
/	(-> (int int) int)	builtin operator
remainder	(-> (int int) int)	builtin operator
modulo	(-> (int int) int)	builtin operator
neg	(-> (int) int)	builtin operator
abs	(-> (int) int)	
null?	(generic (t) (-> ((listof t)) bool))	
null	(generic (t) (-> () (listof t)))	
cons	(generic (t) (-> (t (listof t)) (listof t)))	
car	(generic (t) (-> ((listof t)) t))	
cdr	(generic (t) (-> ((listof t)) (listof t)))	
set-car!	(generic (t) (-> ((listof t) t) unit))	
set-cdr!	(generic (t) (-> ((listof t) (listof t)) unit))	
length	(generic (t) (-> ((listof t)) int))	
append	(generic (t) (-> ((listof t) (listof t)) (listof t)))	

Name	Type	Comment
reverse	(generic (t) (-> ((listof t)) (listof t)))	
list-tail	(generic (t) (-> ((listof t) int) (listof t)))	
list-ref	(generic (t) (-> ((listof t) int) t))	
map	(generic (t1 t2) (-> ((-> (t1) t2) (listof t1)) (listof t2)))	
for-each	(generic (t1 t2) (-> ((-> (t1) t2) (listof t1)) unit))	
reduce	(generic (t1 t2) (-> ((-> (t1 t2) t2) (listof t1) t2) t2))	
list->string	(-> ((listof char)) string)	
string->list	(-> (string) (listof char))	
pair	(generic (t1 t2) (-> (t1 t2) (pairof t1 t2)))	
left	(generic (t1 t2) (-> ((pairof t1 t2)) t1))	
right	(generic (t1 t2) (-> ((pairof t1 t2)) t2))	
ref	(generic (t) (-> (t) (refof t)))	
^	(generic (t) (-> ((refof t)) t))	
:=	(generic (t) (-> ((refof t) t) unit))	
make-string	(-> (int char) string)	
string-length	(-> (string) int)	
string-ref	(-> (string int) char)	
string-set!	(-> (string int char) unit)	
string-fill!	(-> (string char) unit)	
string=?	(-> (string string) bool)	
string<?	(-> (string string) bool)	
string>?	(-> (string string) bool)	
string<=?	(-> (string string) bool)	
string>=?	(-> (string string) bool)	
string-ci=?	(-> (string string) bool)	
string-ci<?	(-> (string string) bool)	
string-ci>?	(-> (string string) bool)	
string-ci<=?	(-> (string string) bool)	
string-ci>=?	(-> (string string) bool)	
substring	(-> (string int int) string)	
string-append	(-> (string string) string)	
string-copy	(-> (string) string)	
string->vector	(-> (string) (vectorof char))	
vector->string	(-> ((vectorof char)) string)	

Name	Type	Comme
sym->string	(-> (sym) string)	
string->sym	(-> (string) sym)	
sym=?	(-> (sym sym) bool)	built
hash	(-> (sym) int)	
make-vector	(generic (t) (-> (int t) (vectorof t)))	
vector-length	(generic (t) (-> ((vectorof t)) int))	
vector-ref	(generic (t) (-> ((vectorof t) int) t))	
vector-set!	(generic (t) (-> ((vectorof t) int t) unit))	
vector-fill!	(generic (t) (-> ((vectorof t) t) unit))	
vector->list	(generic (t) (-> ((vectorof t)) (listof t)))	
list->vector	(generic (t) (-> ((listof t)) (vectorof t)))	
vector-map	(generic (t1 t2) (-> ((-> (t1) t2) (vectorof t1)) (vectorof t2)))	
vector-map2	(generic (t1 t2 t3) (-> ((-> (t1 t2) t3) (vectorof t1) (vectorof t2)) (vectorof t3)))	
vector-reduce	(generic (t1 t2) (-> ((-> (t1 t2) t2) (vectorof t1) t2) t2))	
scan	(generic (t) (-> ((-> (t t) t) (vectorof t)) (vectorof t)))	
segmented-scan	(generic (t) (-> ((-> (t t) t) (vectorof bool) (vectorof t)) (vectorof t)))	
compress	(generic (t1) (-> ((vectorof bool) (vectorof t)) (vectorof t)))	
expand	(generic (t1) (-> ((vectorof bool) (vectorof t) (vectorof t)) (vectorof t)))	
eoshift	(generic (t1) (-> (int (vectorof t) (vectorof t)) (vectorof t)))	

```
Name              Type                                                    Comment
unparse-bool      (-> (bool) string)
unparse-char      (-> (char) string)
unparse-int       (-> (int) string)
unparse-string    (-> (string) string)
unparse-symbol    (-> (sym) string)
unparse-unit      (-> (unit) string)
unparse-list      (generic (t)
                      (-> ((-> (t) string) (listof t)) string))
unparse-vector    (generic (t)
                      (-> ((-> (t) string) (vectorof t)) string))
unparse-pair      (generic (r l)
                      (-> ((-> (l) string)
                             (-> (r) string)
                             (pairof r l))
                           string))
```

C μFX/DLX Compiler Implementation

This appendix contains a snapshot as of February 12, 1992 of the 17 source files which implement the compiler and ocode simulator in Mini-FX. All of these files are available via FTP.

The files included in this appendix are as follows:

Filename	Module	Purpose
compiler/asm.fx	Support	Assemble and unparse ocode representation
compiler/bits.fx	Support	Low-level data manipulation utilities
compiler/dlxsim.fx	Emulate	Emulation support
compiler/exp.fx	Parse	μFX parse tree definition
compiler/exp2ic.fx	Translate	Translate expressions to intermediate code
compiler/ic2oc.fx	Generate	Generate ocode by recursive descent of icode
compiler/icode.fx	Generate	Icode representation definition
compiler/lib.fx	Runtime	Definitions of μFX primitives
compiler/misc.fx	Support	Miscellaneous utilities
compiler/oc2txt.fx	Output	Produce DLX assembly code text
compiler/ocode.fx	Generate	Ocode representation definition
compiler/optimize.fx	Optimize	Optimization of intermediate code
compiler/parse.fx	Parse	Parse μFX Syntax
compiler/system.fx	Emulate	Garbage collector and allocator for ocode emulation
compiler/table.fx	Support	Symbol table utility
compiler/toplevel.fx	Support	Top-level user interface to μFX/DLX
compiler/ty_recon.fx	Reconstruct	Type reconstruction algorithm

The index at the end of this document contains entries for procedures, shared variables, and runtime entry points.

C.1 compiler/asm.fx

The contents of the file compiler/asm.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

;; asm.fx -- assemble and unparse ocode from code generator.  "Assemble"
;; means prepare for emulation; unparse means prepare for output to file
;; for dlxsim assembler.

;; We factor instructions according to the type of operands they use.
;; The following shorthands are used to name these types: i = integer
;; register; c = integer constant; f = single-precision FP; g = double
;; FP; d = integers in FP registers (e.g., for div instruction).

;; The two functions exported by this file are asm-ocode and
;; unparse-ocode.  asm-ocode takes an operation name (symbol) and a
;; set of operands and returns a thunk that, when applied, mutates the
;; state of the DLX machine (see dlxsim.fx) according to the
;; instruction.  unparse-ocode takes an operation name and set of
;; 'rands and returns a string representing the instruction in
;; official DLX assembler syntax.

;; asm-ocode and unparse-ocode use asm-??? (assemble) and unp-???
;; (unparse) functionals for different classes of instructions.  The
;; asm-??? functionals return a thunk which, when applied, mutates the
;; DLX machine state according to the instruction.  The asm-???
;; functionals try to do expensive operations (e.g., matches on
;; operands) in an environment enclosing the returned thunk.  The
;; unp-???  functionals return a string which is the unparsing of the
;; operation.

(define unp-oname
  (lambda (op)
    (let ((oname (down-sym op)))
      (string-append oname (pad oname 8)))))
```

```
(define unparse-reg
  (lambda (rn)
    (if (^ *rr-pretty*)
        (cond ((= rn ZERO) "ZERO")
              ((= rn VAL) "VAL")
              ((= rn ENV) "ENV")
              ((= rn FP) "FP")
              ((= rn SP) "SP")
              ((= rn ARG0) "ARG0")
              ((= rn ARG1) "ARG1")
              ((= rn ARG2) "ARG2")
              ((= rn ARG3) "ARG3")
              ((= rn ARG4) "ARG4")
              ((= rn ARG5) "ARG5")
              ((= rn ARG6) "ARG6")
              ((= rn ARG7) "ARG7")
              ((= rn ARG8) "ARG8")
              ((= rn ATEMP) "ATEMP")
              ((= rn RETADR) "RETADR")
              (else (string-append "r" (int->string rn))))
         (string-append "r" (int->string rn)))))

(define unparse-freg
  (lambda (n) (string-append "f" (int->string n))))

(define unparse-dreg
  (lambda (n) (string-append "f" (int->string n))))

(define asm-iii
  ;; Three integer registers (e.g., add, subu)
  (lambda (fn rands)
    (match rands
     ((rrr->rands^ d s1 s2)
      (lambda () (set-reg! d (fn (get-reg s1) (get-reg s2))))))))

(define unp-iii
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((rrr->rands^ d s1 s2)
        (string-append oname (unparse-reg d) ", " (unparse-reg s1) ", "
                       (unparse-reg s2)))))))

(define asm-iic
  ;; Two int reg's and a constant (e.g., xori)
  (lambda (fn rands)
    (match rands
     ((rri->rands^ d s1 s2)
      (let ((val (eval-immed s2)))
        (lambda () (set-reg! d (fn (get-reg s1) val))))))))
```

```
(define unp-iic
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((rri->rands^ d s1 s2)
        (string-append oname
                       (unparse-reg d) ", " (unparse-reg s1) ", " s2))))))

(define asm-b1
  ;; Branches that take one operand (e.g., jal, bfpf)
  (lambda (fn rands)
    (match rands
     ((i->rands^ l)
      (let ((val (eval-immed l)))
        (lambda () (:= *npc* (fn val)))))
     ((r->rands^ r)
      (lambda () (:= *npc* (fn r)))))))

(define unp-b1
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((i->rands^ l) (string-append oname l))
       ((r->rands^ r) (string-append oname (unparse-reg r)))))))

(define asm-b2
  ;; Branches that take two operands (e.g., bneq)
  (lambda (fn rands)
    (match rands
     ((ri->rands^ r l)
      (let ((val (eval-immed l)))
        (lambda () (:= *npc* (fn r val))))))))

(define unp-b2
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((ri->rands^ r l) (string-append oname (unparse-reg r) ", " l))))))

(define asm-load
  ;; All load instructions except lhi and FP loads
  (lambda (fn rands)
    (match rands
     ((load->rands^ d offset base)
      (lambda ()
        (set-reg! d (fn (get-mem (+ offset (get-reg base))))))))))
```

```
(define unp-load
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((load->rands~ d ofst base)
        (string-append oname
                       (unparse-reg d) ", "
                       (int->string ofst) "(" (unparse-reg base) ")")))))))

(define asm-lhi
  (lambda (rands)
    (match rands
     ((ri->rands~ d s)
      (let ((val (* (eval-immed s) two^16)))
        (lambda () (set-reg! d val)))))))

(define unp-lhi
  (lambda (op rands)
    (match rands
     ((ri->rands~ d s) (string-append "lhi     " (unparse-reg d) ", " s)))))

(define asm-store
  ;; All store instructions except FP stores
  (lambda (fn rands)
    (match rands
     ((store->rands~ offset base v)
        (lambda ()
          (let* ((addr (+ offset (get-reg base)))
                 (old-val (get-mem addr))
                 (val (get-reg v)))
            (set-mem! addr (fn old-val val))))))))

(define unp-store
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((store->rands~ ofst base v)
        (string-append oname
                       (int->string ofst) "(" (unparse-reg base) "), "
                       (unparse-reg v)))))))

(define asm-fff
  ;; Instructions taking three single FP registers (e.g., addf)
  (lambda (fn rands)
    (match rands
     ((rrr->rands~ d s1 s2)
      (lambda () (set-freg! d (fn (get-freg s1) (get-freg s2)))))))))
```

```
(define unp-fff
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((rrr->rands^ d s1 s2)
        (string-append oname (unparse-freg d) ", " (unparse-freg s1) ", "
                       (unparse-freg s2)))))))

(define asm-ggg
  ;; Instructions taking three double FP reg's (e.g., divd)
  (lambda (fn rands)
    (match rands
     ((rrr->rands^ d s1 s2)
      (lambda () (set-dreg! d (fn (get-dreg s1) (get-dreg s2))))))))

(define unp-ggg
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
       ((rrr->rands^ d s1 s2)
        (string-append oname (unparse-dreg d) ", " (unparse-dreg s1) ", "
                       (unparse-dreg s2)))))))

(define asm-fp-rel
  ;; Floating point relational operations modify the fp boolean flag
  (lambda (fn c rands)
    (let ((get (cond ((sym=? c 'f) get-freg) ((sym=? c 'g) get-dreg))))
      (match rands
       ((rr->rands^ s1 s2)
        (lambda () (:= *fp-cond* (fn (get s1) (get s2)))))))))

(define unp-fp-rel
  (lambda (op c rands)
    (let ((unp (cond ((sym=? c 'f) unparse-freg) ((sym=? c 'g) unparse-dreg)))
          (oname (unp-oname op)))
      (match rands
       ((rr->rands^ s1 s2)
        (lambda () (string-append oname (unp s1) (unp s2))))))))

(define asm-cnv
  (let ((get (lambda (c) (cond ((or (sym=? c 'd) (sym=? c 'f)) get-freg)
                              ((sym=? c 'g) get-dreg)
                              ((sym=? c 'i) get-reg))))
        (st! (lambda (c) (cond ((or (sym=? c 'd) (sym=? c 'f)) set-freg!)
                              ((sym=? c 'g) set-dreg!)
                              ((sym=? c 'i) set-reg!)))))
    (lambda (fn d s rands)
      (let ((g (get s)) (s! (st! d)))
        (match rands
         ((rr->rands^ dst src) (lambda () (s! dst (fn (g src))))))))))
```

```
(define unp-cnv
  (let ((unp (lambda (c) (cond ((or (sym=? c 'd) (sym=? c 'f)) unparse-freg)
                               ((sym=? c 'g) unparse-dreg)
                               ((sym=? c 'i) unparse-reg)))))
    (lambda (op d s rands)
      (let ((oname (unp-oname op)))
        (match rands
         ((rr->rands^ dst src)
          (string-append oname ((unp d) dst) ", " ((unp s) src))))))))

;;
;; Functions that are applied to operands (represented as bignums) to
;; execute low-level ALU ops and return the result (another bignum).
;; These functions are used only when the minifx equivilant doesn't do
;; the right thing, such as when when dealing with the inidividual
;; bits of a signed integer.
;;

(define mk-bwise
  ;; Takes an operation (e.g., and?) and applies it bit-wise to the bits
  ;; in the operands.  Since we store the operands in bignum rather
  ;; than machine int format, we need to convert the operands to
  ;; machine ints to make sure negative numbers are handled correctly.
  (lambda (op)
    (lambda (val1 val2)
      (letrec ((loop
                (lambda (x v1 v2 i)
                  (if (= i 32)
                      (mint2bignum x)
                      (loop (+ x (if (op (odd? v1) (odd? v2)) (expt 2 i) 0))
                            (quotient v1 2)
                            (quotient v2 2)
                            (+ i 1))))))
        (loop 0 (bignum2mint val1) (bignum2mint val2) 0)))))

;; For relational operators, must return 1 or 0 rather than #t or #f
(define mk-alu-rel
  (lambda (fn) (lambda (x y) (if (fn x y) 1 0))))

(define mk-alu-urel ;; For unsigned operands
  (lambda (fn) (lambda (x y) (if (fn (bignum2mint x) (bignum2mint y)) 1 0))))

(define mk-alu-sgn ;; For signed integer operators
  (lambda (fn) (lambda (x y) (let ((result (fn x y)))
                               (if (int32? result)
                                   result
                                   (error "ALU overflow"))))))

(define mk-alu-unsgn ;; For unsigned integer operators
  (lambda (fn)
    (lambda (x y)
      (mint2bignum (remainder (fn (bignum2mint x) (bignum2mint y)) two^32)))))

(define alu-neq (lambda (x y) (if (= x y) 0 1)))
```

```
(define alu-xor (mk-bwise (lambda (x y) (or (and x (not y)) (and (not x) y)))))

(define alu-or
  ;; The code generator uses an or instruction with the R0 register as
  ;; moves, so we want that case to go fast.
  (lambda (x y)
    (cond ((= x 0) y)
          ((= y 0) x)
          (else ((mk-bwise or?) x y)))))

(define alu-s-left-l (lambda (x y) (* x (expt 2 y))))

(define alu-s-right-l
  (lambda (x y) (mint2bignum (quotient (bignum2mint x) (expt 2 y)))))

(define alu-s-right-a
  (lambda (x y)
    (mint2bignum (+ (quotient (bignum2mint x) (expt 2 y))
                    (if (< x 0) two^31 0)))))
```

```
(define asm
  (lambda (insn)
    (let ((op (op-code insn))
          (rands (op-rands insn)))
      (cond
       ((sym=? op 'add)   (asm-iii (mk-alu-sgn +) rands))
       ((sym=? op 'addi)  (asm-iic (mk-alu-sgn +) rands))
       ((sym=? op 'addu)  (asm-iii (mk-alu-unsgn +) rands))
       ((sym=? op 'addui) (asm-iic (mk-alu-unsgn +) rands))
       ((sym=? op 'and)   (asm-iii (mk-bwise and?) rands))
       ((sym=? op 'andi)  (asm-iic (mk-bwise and?) rands))
       ((sym=? op 'or)    (asm-iii alu-or rands))
       ((sym=? op 'ori)   (asm-iic alu-or rands))
       ((sym=? op 'seq)   (asm-iii (mk-alu-rel =) rands))
       ((sym=? op 'seqi)  (asm-iic (mk-alu-rel =) rands))
       ((sym=? op 'sequ)  (asm-iii (mk-alu-urel =) rands))
       ((sym=? op 'sequi) (asm-iic (mk-alu-urel =) rands))
       ((sym=? op 'sge)   (asm-iii (mk-alu-rel >=) rands))
       ((sym=? op 'sgei)  (asm-iic (mk-alu-rel >=) rands))
       ((sym=? op 'sgeu)  (asm-iii (mk-alu-urel >=) rands))
       ((sym=? op 'sgeui) (asm-iic (mk-alu-urel >=) rands))
       ((sym=? op 'sgt)   (asm-iii (mk-alu-rel >) rands))
       ((sym=? op 'sgti)  (asm-iic (mk-alu-rel >) rands))
       ((sym=? op 'sgtu)  (asm-iii (mk-alu-urel >) rands))
       ((sym=? op 'sgtui) (asm-iic (mk-alu-urel >) rands))
       ((sym=? op 'sle)   (asm-iii (mk-alu-rel <=) rands))
       ((sym=? op 'slei)  (asm-iic (mk-alu-rel <=) rands))
       ((sym=? op 'sleu)  (asm-iii (mk-alu-urel <=) rands))
       ((sym=? op 'sleui) (asm-iic (mk-alu-urel <=) rands))
       ((sym=? op 'sll)   (asm-iii alu-s-left-l rands))
       ((sym=? op 'slli)  (asm-iic alu-s-left-l rands))
       ((sym=? op 'slt)   (asm-iii (mk-alu-rel <) rands))
       ((sym=? op 'slti)  (asm-iic (mk-alu-rel <) rands))
       ((sym=? op 'sltu)  (asm-iii (mk-alu-urel <) rands))
       ((sym=? op 'sltui) (asm-iic (mk-alu-urel <) rands))
       ((sym=? op 'sne)   (asm-iii alu-neq rands))
       ((sym=? op 'snei)  (asm-iic alu-neq rands))
       ((sym=? op 'sneu)  (asm-iii alu-neq rands))
       ((sym=? op 'sneui) (asm-iic alu-neq rands))
       ((sym=? op 'sra)   (asm-iii alu-s-right-a rands))
       ((sym=? op 'srai)  (asm-iic alu-s-right-a rands))
       ((sym=? op 'srl)   (asm-iii alu-s-right-l rands))
       ((sym=? op 'srli)  (asm-iic alu-s-right-l rands))
       ((sym=? op 'sub)   (asm-iii (mk-alu-sgn -) rands))
       ((sym=? op 'subi)  (asm-iic (mk-alu-sgn -) rands))
       ((sym=? op 'subu)  (asm-iii (mk-alu-unsgn -) rands))
       ((sym=? op 'subui) (asm-iic (mk-alu-unsn -) rands))
       ((sym=? op 'xor)   (asm-iii alu-xor rands))
       ((sym=? op 'xori)  (asm-iic alu-xor rands))
```

```
((sym=? op 'beqz)  (asm-b2 (lambda (r d)
                             (if (= (get-reg r) 0) d (^ *npc*)))
                           rands))
((sym=? op 'bfpt)  (asm-b1 (lambda (d) (if (^ *fp-cond*) (^ *npc*) d))
                           rands))
((sym=? op 'bfpf)  (asm-b1 (lambda (d) (if (^ *fp-cond*) d (^ *npc*)))
                           rands))
((sym=? op 'bnez)  (asm-b2 (lambda (r d)
                             (if (= (get-reg r) 0) (^ *npc*) d))
                           rands))
((sym=? op 'j)     (asm-b1 id rands))
((sym=? op 'jr)    (asm-b1 get-reg rands))
((sym=? op 'jal)   (asm-b1 (lambda (d)
                             (begin (set-reg! RETADR (+ (^ *pc*) 4))
                                    d))
                           rands))
((sym=? op 'jalr)  (asm-b1 (lambda (r)
                             (begin (set-reg! RETADR (+ (^ *pc*) 4))
                                    (get-reg r)))
                           rands))

((sym=? op 'lb)    (asm-load (lambda (v) (sext8->32 (remainder v two^8)))
                             rands))
((sym=? op 'lbu)   (asm-load (lambda (v) (remainder v two^8)) rands))
((sym=? op 'lh)    (asm-load (lambda (v)
                               (sext16->32 (remainder v two^16)))
                             rands))
((sym=? op 'lhi)   (asm-lhi rands))
((sym=? op 'lhu)   (asm-load (lambda (v) (remainder v two^16)) rands))
((sym=? op 'lw)    (asm-load id rands))

((sym=? op 'sb)    (asm-store
                    (let ((fn (set-bit-field 0 8)))
                      (lambda (ov v) (fn ov (remainder v two^8))))
                    rands))
((sym=? op 'sh)    (asm-store
                    (let ((fn (set-bit-field 0 16)))
                      (lambda (ov v) (fn ov (remainder v two^16))))
                    rands))
((sym=? op 'sw)    (asm-store (lambda (ov v) v) rands))

((sym=? op 'div)   (asm-fff quotient rands))
((sym=? op 'divu)  no-emulate)
((sym=? op 'mult)  (asm-fff (mk-alu-sgn *) rands))
((sym=? op 'multu) no-emulate)
```

```
        ((sym=? op 'addd)  (asm-ddd fl+ rands))
        ((sym=? op 'addf)  (asm-fff fl+ rands))
        ((sym=? op 'divd)  (asm-ddd fl/ rands))
        ((sym=? op 'divf)  (asm-fff fl/ rands))
        ((sym=? op 'eqd)   (asm-fp-rel 'g fl= rands))
        ((sym=? op 'eqf)   (asm-fp-rel 'f fl= rands))
        ((sym=? op 'ged)   (asm-fp-rel 'g fl>= rands))
        ((sym=? op 'gef)   (asm-fp-rel 'f fl>= rands))
        ((sym=? op 'gtd)   (asm-fp-rel 'g fl> rands))
        ((sym=? op 'gtf)   (asm-fp-rel 'f fl> rands))
        ((sym=? op 'ltd)   (asm-fp-rel 'g fl< rands))
        ((sym=? op 'ltf)   (asm-fp-rel 'f fl< rands))
        ((sym=? op 'led)   (asm-fp-rel 'g fl<= rands))
        ((sym=? op 'lef)   (asm-fp-rel 'f fl<= rands))
        ((sym=? op 'multd) (asm-ddd fl* rands))
        ((sym=? op 'multf) (asm-fff fl* rands))
        ((sym=? op 'ned)   (asm-fp-rel 'g (lambda (x y) (not (fl= x y))) rands))
        ((sym=? op 'nef)   (asm-fp-rel 'f (lambda (x y) (not (fl= x y))) rands))
        ((sym=? op 'subd)  (asm-ddd fl- rands))
        ((sym=? op 'subf)  (asm-fff fl- rands))

        ((sym=? op 'ld)    no-emulate)
        ((sym=? op 'lf)    no-emulate)
        ((sym=? op 'sd)    no-emulate)
        ((sym=? op 'sf)    no-emulate)

        ((sym=? op 'trap)  no-emulate)
        ((sym=? op 'nop)   (lambda () the-unit))

        ((sym=? op 'cvtd2f)  (asm-cnv id 'f 'g rands))
        ((sym=? op 'cvtd2i)  (asm-cnv id 'd 'g rands))
        ((sym=? op 'cvtf2d)  (asm-cnv id 'g 'f rands))
        ((sym=? op 'cvtf2i)  (asm-cnv truncate 'd 'f rands))
        ((sym=? op 'cvti2d)  (asm-cnv int->float 'g 'd rands))
        ((sym=? op 'cvti2f)  (asm-cnv int->float 'f 'd rands))
        ((sym=? op 'movd)    (asm-cnv id 'g 'g rands))
        ((sym=? op 'movf)    (asm-cnv id 'f 'f rands))
        ((sym=? op 'movfp2i) (asm-cnv id 'i 'd rands))
        ((sym=? op 'movi2fp) (asm-cnv id 'd 'i rands))))))

;;   (asm-info 'movi2s "0x00" "0x30")
;;   (asm-info 'movs2i "0x00" "0x31")
;;    (asm-info 'rfe "0x40" "0x00")
```

```
(define unparse-ocode
  (lambda (insn)
    (let ((op (op-code insn))
          (rands (op-rands insn)))
      (cond
       ((sym=? op 'labeldef)
        (string-append (match rands ((label->rands~ l) l)) ":"))
       ((sym=? op 'stringdef)
        (string-append ".asciiz \"" (match rands ((string->rands~ s) s)) "\""
                       (char->string #\newline) ".align 2"))
       ((sym=? op 'worddef)
        (string-append ".word    "
                       (int->string (match rands ((word->rands~ w) w)))))

       ((memq op '(add addu and or seq sequ sge sgeu sgt sgtu sle sleu
                   sll slt sltu sne sneu sra srl sub subu xor))
        (unp-iii op rands))

       ((memq op '(addi addui andi ori seqi sequi sgei sgeui sgti sgtui slei
               sleui slli slti sltui snei sneui srai srli subi subui xori))
        (unp-iic op rands))

       ((memq op '(beqz bnez))                 (unp-b2 op rands))
       ((memq op '(bfpt bfpf bnez j jr jal jalr)) (unp-b1 op rands))
       ((memq op '(lb lbu lh lhu lw))          (unp-load op rands))
       ((sym=? op 'lhi)                        (unp-lhi op rands))
       ((memq op '(sb sh sw))                  (unp-store op rands))
       ((memq op '(div divu mult multu))       (unp-fff op rands))
       ((memq op '(addf divf multf subf))      (unp-fff op rands))
       ((memq op '(addd divd multd subd))      (unp-ddd op rands))
       ((memq op '(eqf gef gtf ltf lef nef)) (unp-fp-rel 'f op rands))
       ((memq op '(eqd ged gtd ltd led ned)) (unp-fp-rel 'g op rands))

       ((sym=? op 'nop)     "nop")
       ((sym=? op 'cvtd2f)  (unp-cnv op 'f 'g rands))
       ((sym=? op 'cvtd2i)  (unp-cnv op 'd 'g rands))
       ((sym=? op 'cvtf2d)  (unp-cnv op 'g 'f rands))
       ((sym=? op 'cvtf2i)  (unp-cnv op 'd 'f rands))
       ((sym=? op 'cvti2d)  (unp-cnv op 'g 'd rands))
       ((sym=? op 'cvti2f)  (unp-cnv op 'f 'd rands))
       ((sym=? op 'movd)    (unp-cnv op 'g 'g rands))
       ((sym=? op 'movf)    (unp-cnv op 'f 'f rands))
       ((sym=? op 'movfp2i) (unp-cnv op 'i 'd rands))
       ((sym=? op 'movi2fp) (unp-cnv op 'd 'i rands))))))

     ;; ((memq op '(ld lf sd sf)) ??)
     ;; ((sym=? op 'trap) ??)
```

C.2 compiler/bits.fx

The contents of the file compiler/bits.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-
```

```
;;
;; Bit banging stuff
;;

(define two^8 (expt 2 8))
(define two^15 (expt 2 15))
(define two^16 (expt 2 16))
(define two^31 (expt 2 31))
(define two^32 (expt 2 32))

(define int16? ; (-> (int) bool)
  (lambda (n) (and (<= n (- two^15 1)) (>= n (- 0 two^15)))))

(define int32?
  (lambda (n) (and (<= n (- two^31 1)) (>= n (- 0 two^31)))))



(define mk-bignum2mint
  (lambda (size-in-bits)
    (let ((two^n (expt 2 size-in-bits)))
      (lambda (bn) (if (>= bn 0) bn (+ two^n bn))))))
(define bignum2mint (mk-bignum2mint 32))

(define mk-bignum2bits
  (lambda (size-in-bits)
    (letrec ((loop
              (lambda (s v i)
                (if (= i size-in-bits)
                    s
                    (loop (string-append (if (odd? v) "1" "0") s)
                          (quotient v 2)
                          (+ i 1))))))
      (lambda (bn) (loop "" (bignum2signed bn) 0)))))
(define bignum2bits (mk-bignum2bits 32))

(define mk-mint2bignum
  (lambda (size-in-bits)
    (let ((two^n (expt 2 size-in-bits))
          (two^n-1 (expt 2 (- size-in-bits 1))))
      (lambda (mi)
        (if (<= mi (- two^n-1 1)) mi (- mi two^n))))))
(define mint2bignum (mk-mint2bignum 32))



(define get-bit-field
  ;; bit-field : int * int -> (int -> int)
  ;; Access a bit field of an integer.
  (lambda (position size)
    (let ((shift (expt 2 position))
          (mask (expt 2 size)))
      (lambda (n)
        (remainder (quotient n shift) mask)))))
```

```
(define set-bit-field
  ;; set-bit-field : int * int -> (int * int -> int)
  ;; Initialize a bit field of an integer.  Assumes the field currently
  ;; contains zero, because we don't need to clear it first in that case.
  (lambda (position size)
    (let ((shift (expt 2 position))
          (mask (expt 2 size)))
      (lambda (n value)
        (if (or (< value 0)
                (>= value mask))
            (error "bit field out of range" position size n value)
            (+ n (* value shift)))))))

;;
;; Parse and unparse hex numbers to make examination of bit-operations
;; easier.
;;

(define h2i
  ;; Convert string of hex numbers into integer.  Requires string to
  ;; be in form "0xh..." where h is in [0-9a-z]
  (letrec ((loop
            (lambda (s i v)
              (if (= i (string-length s))
                  v
                  (let* ((c (string-ref s i))
                         (h (if (char-numeric? c)
                                (- (char->int c) (char->int #\0))
                                (+ (- (char->int c) (char->int #\a)) 10))))
                    (loop s (+ i 1) (+ (* v 16) h)))))))
    (lambda (s) (loop s 2 0))))

(define i2h
  (letrec ((loop
            (lambda (v l)
              (if (= v 0)
                  l
                  (let* ((h (remainder v 16))
                         (c (if (<= h 9)
                                (int->char (+ h (char->int #\0)))
                                (int->char (+ (- h 10) (char->int #\a))))))
                    (loop (quotient v 16) (cons c l)))))))
    (lambda (v)
      (if (= v 0)
          "0x0"
          (list->string (cons #\0 (cons #\x (loop v (null)))))))))
```

C.3 compiler/dlxsim.fx

The contents of the file compiler/dlxsim.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-
```

```
;; Emulator parameters
(define *semispace-size* (ref 500))           ; Slots in each space
(define *stack-size* (ref 500))               ; Slots in stack
(define *noisy-gc* (ref #t))
(define *rr-pretty* (ref #t))                 ; Pretty-print registers
(define *program-start* (ref 0))              ; Addr of 1st word of code
(define *break-points* (ref (list)))

;; Emulator state
(define *entire-memory-size* (ref 0))         ; Total words in sim'tor mem
(define *mem* (ref (make-vector 0)))
(define *end-program* (ref 0))                ; Addr of last word of code
(define *this-semispace* (ref 0))             ; (Tagged) address of first
(define *other-semispace* (ref 0))            ;   word in space
(define *this-semispace-end* (ref 0))         ; (T'd) address of 1st word
(define *other-semispace-end* (ref 0))        ;   beyond last word in space

(define *reg* (generate-vector 32 (lambda (index) 0)))
(define *freg* (generate-vector 32 (lambda (index) 0.0)))
(define *fp-cond* (ref 0))
(define *pc* (ref 0))
(define *npc* (ref 0))
(define *halt-emulate?* (ref #f))
(define *label-table* (ref (null)))

;; Emulator statistics
(define *instruction-count* (ref 0))
(define *num-gcs* (ref 0))
(define *gc-words-copied* (ref 0))
(define *total-allocation* (ref 0))
(define *total-allocs* (ref 0))
(define *max-stack-size* (ref 0))

;;
;;  Routines to set up emulator
;;

;; Initialize emulator
(define init-emulator
  (lambda ()
    (begin
      ;; First pass of assembler: get name of labels and find out length of
      ;; program code.
      (:= *label-table* (null))
      (enter-system-routine-labels) ; This is how we call system routines.
      (asm-pass1 (^ ocode-list))    ; Calculate values of labels

      ;; Calculate needed memory size
      (:= *entire-memory-size* (+ (quotient (^ *end-program*) 4)
                                  (* 2 (^ *semispace-size*))
                                  (^ *stack-size*)))
```

```
    ;; Create a vector to represent the DLX machine's memory.  If required
    ;; memory size hasn't grown much since last time, then don't cons up
    ;; the new vector so we avoid generating garbage.
    (let ((old-len (vector-length (^ *mem*))))
      (if (or (< old-len (^ *entire-memory-size*))
              (> (- old-len 500) (^ *entire-memory-size*)))
          (:= *mem*
              (generate-vector (^ *entire-memory-size*) (lambda (i) 0)))
          (:= *entire-memory-size* old-len)))

    ;; Now that we have a memory vector, assemble code into that vector
    (asm-pass2 (^ ocode-list))))) ; Assmbl emulator thunks in2 mem vec

(define restart-emulator
  (lambda ()
    (begin
      ;; Zero out regisiters
      (letrec ((loop (lambda (i) (if (>= i 32)
                                     the-unit
                                     (begin (set-reg! i 0)
                                            (set-freg! i 0.0)
                                            (loop (+ i 1)))))))
        (loop 0))

      ;; Initialize important system registers
      (set-reg! SP (- (* (^ *entire-memory-size*) 4) 4))
      (set-reg! HP (+ (^ *end-program*) 1))
      (goto (^ *program-start*)) ; Set pc registers
      (:= *halt-emulate?* #f)

      ;; Set up heap
      (:= *this-semispace* (get-reg HP))
      (:= *this-semispace-end* (+ (get-reg HP) (* (^ *semispace-size*) 4)))
      (:= *other-semispace* (^ *this-semispace-end*))
      (:= *other-semispace-end*
          (+ (^ *other-semispace*) (* (^ *semispace-size*) 4)))

      ;; Clear statistics variables
      (:= *num-gcs* 0)
      (:= *gc-words-copied* 0)
      (:= *total-allocation* 0)
      (:= *total-allocs* 0)
      (:= *instruction-count* 0)
      (:= *max-stack-size* 0)

      ;; Set-up an initial frame to return through
      (set-reg! ATEMP FrameSize)
      (allocate-block-of-memory)
      (set-reg! FP (get-reg ATEMP))
      (set-slot! (get-reg FP) 0 0)
      (set-slot! (get-reg FP) 1 SP)
      (set-slot! (get-reg FP) 2 -16) ; Return to __EXIT system routine
      (set-slot! (get-reg FP) 3 0)
      (save-regs-into-frame)
```

```
        ;; Set-up an inital closure for calling START_1
        (set-reg! ATEMP 4)        ; 2 words (tagged)
        (allocate-block-of-memory)
        (set-reg! ARGO (get-reg ATEMP)) ; Create dummy closure
        (set-slot! (get-reg ARGO) 0 (^ *program-start*))
        (set-slot! (get-reg ARGO) 1 0))))

;;
;; Routines to do emulations after machine has been set up
;;

(define emulate
  (lambda ()
    (begin (emulate-one-instruction)
           (if (^ *halt-emulate?*) the-unit (emulate)))))

(define rerun
  ;;  rerun the compiled program from the top...
  (lambda () (begin (restart-emulator) (emulate))))

(define step
  ;; Single-step program, printing out instruction just emulated.
  (lambda ()
    (let ((temp (^ *verbose-flag*)))
      (begin (:= *verbose-flag* #f)
             (display-one-instruction (^ *pc*))
             (nstep 1)
             (display-one-instruction (^ *pc*))
             (:= *verbose-flag* temp)))))

(define nstep
  (lambda (n)
    (letrec ((loop (lambda (i)
                     (cond ((>= i n) the-unit)
                           ((^ *halt-emulate?*)
                            (begin (display "Execution terminated.")
                                   the-unit))
                           (else (begin (emulate-one-instruction)
                                        (loop (+ i 1))))))))
      (begin (loop 0) (dump)))))

;;
;; Two passes of assembler
;;
```

```
(define asm-pass1
  ;; Calculate values of labels and enter them into symbol table.  Set
  ;; *end-program* to (untagged) address of 1st word after program code.
  (lambda (ocode)
    (letrec ((loop (lambda (ocode pc)
                     (if (null? ocode)
                         pc
                         (match (car ocode)
                          ((ocode^ 'labeldef (label->rands^ label))
                           (begin (enter-label label pc)
                                  (loop (cdr ocode) pc)))
                          ((ocode^ 'stringdef _) (loop (cdr ocode) (+ pc 4)))
                          (_ (loop (cdr ocode)   (+ pc 4))))))))
      (:= *end-program* (loop ocode (^ *program-start*))))))

(define asm-pass2
  ;; Place emulation thunks into memory
  (lambda (ocode)
    (letrec ((loop (lambda (ocode pc)
                     (if (null? ocode)
                         the-unit
                         (match (car ocode)
                          ((ocode^ 'labeldef _) (loop (cdr ocode) pc))
                          ((ocode^ 'stringdef (string->rands^ string))
                           (begin (set-mem! pc string)
                                  (loop (cdr ocode) (+ pc 4))))
                          ((ocode^ 'worddef (word->rands^ word))
                           (begin (set-mem! pc word)
                                  (loop (cdr ocode) (+ pc 4))))
                          (_
                           (begin (set-mem! pc (asm (car ocode)))
                                  (loop (cdr ocode) (+ pc 4)))))))))
      (loop ocode 0))))

;;
;; Symbol table for labels -- set up for two-way mapping
;;

(define enter-label
  (lambda (l v) (:= *label-table* (cons (tuple l v) (^ *label-table*)))))

(define label2num
  (lambda (l)
    (letrec ((loop (lambda (lst)
                     (cond ((null? lst) (error "Unknown label" l))
                           ((string=? l (tuple-ref (car lst) 0))
                            (tuple-ref (car lst) 1))
                           (else (loop (cdr lst)))))))
      (loop (^ *label-table*)))))
```

```
(define num2label
  (lambda (n)
    (letrec ((loop (lambda (lst)
                     (cond ((null? lst) "")
                           ((= n (tuple-ref (car lst) 1))
                            (tuple-ref (car lst) 0))
                           (else  (loop (cdr lst)))))))
      (loop (^ *label-table*)))))

;;
;; Routines to change state of emulator according to instructions
;;

(define goto
  (lambda (new-pc)
    (begin
      (if (memq new-pc (^ *break-points*))
          (error "Break-point: type (proceede) to continue."))
      (:= *pc* new-pc)
      (:= *npc* (+ new-pc 4)))))

(define emulate-one-instruction
  (lambda ()
    (begin
      (:= *instruction-count* (+ (^ *instruction-count*) 1))
      (if (^ *verbose-flag*)
          (display-one-instruction (^ *pc*))
          the-unit)
      (if (< (^ *pc*) 0)
          (begin
            ;; fake system subroutines have negative addresses...
            ;; they're also always called with jal's (retaddr in r31)
            (system-routine (^ *pc*))   ; call the service
            (goto (get-reg RETADR)))            ; fake the return.
          (let ((old-pc (^ *pc*)))
            (begin (goto (^ *npc*))
                   ((get-mem old-pc)))))))))

;;
;; Procedures to access and change emulator state
;;

(define get-mem
  (lambda (address)
    (if (= (remainder address 4) 0)
        (vector-ref (^ *mem*) (quotient address 4))
        (error "Unaligned read."))))

(define set-mem!
  (lambda (address val)
    (if (= (remainder address 4) 0)
        (vector-set! (^ *mem*) (quotient address 4) val)
        (error "Unaligned write."))))
```

```
(define get-slot (lambda (ptr slot) (get-mem (+ ptr (otag slot)))))
(define set-slot! (lambda (ptr slot v) (set-mem! (+ ptr (otag slot)) v)))

(define get-reg   (lambda (regnum) (vector-ref *reg* regnum)))
(define set-reg!  (lambda (regnum val) (vector-set! *reg* regnum val)))
(define get-freg  (lambda (fregnum) (vector-ref *freg* fregnum)))
(define set-freg! (lambda (fregnum val) (vector-set! *freg* fregnum val)))

(define get-dreg
  (lambda (fregnum)
    (if (odd? fregnum)
        (error "Bad double register" fregnum)
        (vector-ref *freg* fregnum))))

(define set-dreg!
  (lambda (fregnum val)
    (if (odd? fregnum)
        (error "Bad double register" fregnum)
        (begin (vector-set! *freg* fregnum val)
               (vector-set! *freg* (+ fregnum 1) -3.21)))))

;; Routines to inspect the state of the emulator
(define dump
  ;; Display register values
  (lambda ()
    (letrec ((cols 4) (width 10)
             (loop (lambda (i)
                     (if (>= i 32)
                         the-unit
                         (let ((rval (i2h (get-reg i))))
                           (begin
                             (display (pad rval width)) (display rval)
                             (if (= (remainder i cols) (- cols 1))
                                 (newline)
                                 the-unit)
                             (loop (+ i 1)))))))))
      (begin (newline) (loop 0)))))

(define find-insn
  (lambda (ocode pc)
    (cond ((sym=? (op-code (car ocode)) 'labeldef)
           (find-insn (cdr ocode) pc))
          ((<= pc 0) (car ocode))
          ((sym=? (op-code (car ocode)) 'stringdef)
           (find-insn (cdr ocode) (- pc 4)))
          (else (find-insn (cdr ocode) (- pc 4))))))
```

```
(define display-one-instruction
  (lambda (pc-i)
    (if (< pc-i 0)
        (begin (display "**** ") (display (num2label pc-i)) (newline))
        (let ((l (num2label pc-i)))
          (begin
            (newline)
            (if (> (string-length l) 0)
                (begin (display "     ") (display l) (display ":") (newline))
                the-unit)
            (if (< pc-i 1000) (display "0") the-unit)
            (if (< pc-i 100) (display "0") the-unit)
            (if (< pc-i 10) (display "0") the-unit)
            (display pc-i) (display "  ")
            (display (unparse-ocode (find-insn (^ ocode-list) pc-i))))))))

(define disasm ;;  print program in memory (to see if it's uncorrupted)
  (lambda ()
    (letrec ((loop (lambda (i) (if (>= i (^ *end-program*))
                                   the-unit
                                   (begin (display-one-instruction i)
                                          (loop (+ i 4)))))))
      (loop (^ *program-start*)))))

(define pb ;; print heap block
  (lambda (p)
    (letrec ((blksize (de-itag (get-slot p -1)))
             (loop (lambda (i)
                     (if (>= i blksize)
                         the-unit
                         (begin
                           (display (string-append
                                     "  +" (pad (int->string i) 3) ": "))
                           (display (get-slot p i))
                           (newline)
                           (loop (+ i 1)))))))
      (begin
        (display "Pointer=") (display p)
        (display "  Blocksize=") (display blksize) (newline)
        (loop 0)))))
```

```
;;
;; Evaluate constant expressions in operands of insn's
;;
(define eval-immed
  ;;  ...this is *very* crude, but sufficient for now...
  ;; Accept the following grammar:
  ;;   expr:
  ;;     number
  ;;     -number
  ;;     LABEL
  ;;     LABEL + 1
  ;;     (expr) & 0xffff
  ;;     (expr)>>16
  ;; Very little error checking is done...
  (lambda (s) (av-parse-expr (ref (av-tokenize s)))))

(define av-parse-expr
  (lambda (toks)
    (let ((c (string-ref (car (^ toks)) 0)))
      (cond
       ((char=? c #\( )   ;; (expr) ...
        (begin (:= toks (cdr (^ toks)))
               (mint2bignum (avpe-recurse toks))))
       ((char-numeric? c) ;; number
        (let ((n (av-parse-number (car (^ toks)))))
          (begin (:= toks (cdr (^ toks)))
                 n)))
       ((char=? c #\-)    ;; -number
        (let ((n (- (av-parse-number (cadr (^ toks))))))
          (begin (:= toks (list-tail (^ toks) 2))
                 n)))
       (else              ;; LABEL | LABEL + 1
        (let ((n (label2num (car (^ toks)))))
          (if (and (>= (length (^ toks)) 3)
                   (char=? (string-ref (cadr (^ toks)) 0) #\+)) ;; LABEL + 1
              (begin (:= toks (list-tail (^ toks) 3))
                     (+ 1 n))
              (begin (:= toks (cdr (^ toks)))
                     n))))))))

(define avpe-recurse
  (lambda (toks)
    (let ((n (bignum2mint (av-parse-expr toks))))
      (if (not (char=? (string-ref (car (^ toks)) 0) #\)))
          (error "invalid av-expr syntax")
          (cond ((char=? (string-ref (cadr (^ toks)) 0) #\&)
                 (begin (:= toks (list-tail (^ toks) 3))
                        (modulo n 65536)))
                ((char=? (string-ref (cadr (^ toks)) 0) #\>)
                 (begin (:= toks (list-tail (^ toks) 3))
                        (quotient n 65536)))
                (else (error "invalid av-expr syntax"))))))))
```

```
(define av-parse-number
  ;; Assume we got a decimal number
  (lambda (s)
    (letrec ((loop
              (lambda (n i)
                (if (= i (string-length s))
                    n
                    (loop (+ (* 10 n)
                             (- (char->int (string-ref s i)) (char->int #\0)))
                          (+ i 1))))))
      (loop 0 0))))

;; A crude lexer for a crude parser...

(define av-tokenize
  ;; Real simple lexer for arithmatic expressions in assembly code
  (lambda (s)
    (letrec ((loop
              (lambda (s i)
                (let ((j (av-kill-whitespace s i)))
                  (if (>= j (string-length s))
                      (null)
                      (let ((k (av-next-tok s j)))
                        (cons (substring s j k)
                              (loop s k))))))))
      (loop s 0))))

(define av-kill-whitespace
  (lambda (s i)
    (if (and (< i (string-length s)) (char-whitespace? (string-ref s i)))
        (av-kill-whitespace s (+ i 1))
        i)))

(define av-next-tok
  (lambda (s i)
    (let ((c (string-ref s i))
          (len (string-length s)))
      (cond
       ((char=? c #\() (+ i 1))
       ((char=? c #\)) (+ i 1))
       ((av-sym? c)
        (letrec ((loop (lambda (i)
                         (if (or (= i len) (not (av-sym? (string-ref s i))))
                             i
                             (loop (+ i 1))))))
          (loop (+ i 1))))
       ((av-punct? c)
        (letrec ((loop (lambda (i)
                         (if (or (= i len) (not (av-punct? (string-ref s i))))
                             i
                             (loop (+ i 1))))))
          (loop (+ i 1))))))))
```

```
(define av-sym?
  (lambda (c) (or (char-alphabetic? c) (char-numeric? c) (char=? c #\_))))

(define av-punct?
  (lambda (c) (and (not (av-sym? c)) (not (char=? c #\())
                   (not (char=? c #\))) (not (char-whitespace? c)))))
```

C.4 compiler/exp.fx

The contents of the file compiler/exp.fx:

```
; -*- Mode: Scheme; Package: SCHEME; -*-

; Data types for use by the front-end of the compiler
;       (parser, type reconstructor, exp2ic)

; Expressions

(define-datatype exp                                         ; E ::=
  (variable->exp (refof type) sym)                           ;   I
  (bool->exp (refof type) bool)                              ; | Bool
  (char->exp (refof type) char)                              ; | Char
  (int->exp (refof type) int)                                ; | Int
  (string->exp (refof type) string)                          ; | String
  (sym->exp (refof type) sym)                                ; | (symbol Sym)
  (conditional->exp (refof type) exp exp exp)                ; | (if E1 E2 E3)
  (begin->exp (refof type) (listof exp))                     ; | (begin E1 ...)
  (abstraction->exp (refof type) (listof sym) exp)           ; | (lambda (I*) E)
  (combination->exp (refof type) exp (listof exp))           ; | (E0 E*)
  (binder->exp (refof type) (listof definition) exp)         ; | (let ((I E)*) E)
  (recursion->exp (refof type) (listof definition) exp)      ; | (letrec ((I E)*) E)
  )

(define expression-type
  (lambda (exp)
    (match exp
      ((variable->exp~ type-ptr _) (^ type-ptr))
      ((bool->exp~ _ _) boolean-type)
      ((char->exp~ _ _) character-type)
      ((int->exp~ _ _) integer-type)
      ((string->exp~ _ _) string-type)
      ((sym->exp~ _ _) symbol-type)
      ((conditional->exp~ type-ptr _ _ _) (^ type-ptr))
      ((begin->exp~ type-ptr _) (^ type-ptr))
      ((abstraction->exp~ type-ptr _ _) (^ type-ptr))
      ((combination->exp~ type-ptr _ _) (^ type-ptr))
      ((binder->exp~ type-ptr _ _) (^ type-ptr))
      ((recursion->exp~ type-ptr _ _) (^ type-ptr))
      )))


(define-datatype definition     ; (I E)
  (make-definition sym exp))
```

```
(define (definition-name d)
  (match d
    ((make-definition~ name value) name)))

(define (definition-value d)
  (match d
    ((make-definition~ name value) value)))


; Types

(define-datatype type
  (base->type sym)              ; base type (bool, int, string, symbol, unit)
  (tvariable->type tvariable)   ; type variable
  (compound->type sym           ; ->, listof, etc.
                  (listof type))
  (unknown->type)               ; marking yet unconstrained type variables
  )

(define boolean-type (base->type (symbol bool)))
(define character-type (base->type (symbol char)))
(define integer-type (base->type (symbol int)))
(define string-type  (base->type (symbol string)))
(define symbol-type  (base->type (symbol sym)))
(define unit-type    (base->type (symbol unit)))
(define unknown-type (unknown->type))

(define same-constructor? eq?)

(define arrow-constructor (symbol ->))

(define make-arrow-type
  (lambda (arg-types body-type)
    (compound->type arrow-constructor (cons body-type arg-types))))

(define arrow-takes
  (lambda (ty)
    (match ty
      ((compound->type~ '-> (cons~ bt at)) at))))

(define arrow-returns
  (lambda (ty)
    (match ty
      ((compound->type~ '-> (cons~ bt at)) bt))))

(define is-arrow-type?
  (lambda (ty)
    (match ty
      ((compound->type~ '-> _) #t)
      (_ #f))))


; Type names are symbols
```

```
(define same-name? sym=?)


; Type schemas

(define-datatype schema
  (make-schema (listof tvariable) type))

(define-datatype tvar-or-schema
  (tvar->tvar-or-schema tvariable)
  (schema->tvar-or-schema schema))

(define (schema-generics s)
  (match s
    ((make-schema~ generics typ) generics)))

(define (schema-type s)
  (match s
    ((make-schema~ generics typ) typ)))


; Unit value

(define the-unit (:= (ref 0) 0))
```

C.5 compiler/exp2ic.fx

The contents of the file compiler/exp2ic.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

;; exp2ic.fx -- convert expr's to icode.

;; Depth-first, syntax-driven translation of expressions into
;; intermediate code.  The depth-first translation is stack hungry,
;; but we don't expect extremely deep expressions.

;; The resulting intermediate code is stored on the list "icode-list."
;; The order of the icode on this list is preserved in the ultimate
;; machine code, i.e., the order of the list represents flow of
;; control through the icode.

;; One tricky part is how to generate code for lambda's and letrec's
;; because the code for the body does not belong in-line with the call
;; to the body.  To handle this, we have a list called
;; "icode-to-be-emitted" which is a list of blocks of icode to be
;; emitted later.  When a lambda body is compiled, the icode for the
;; body is put on the icode-to-be-emitted list, while the icode for
;; the closure is put into the current block of icode.


(define icode-list (ref (null))) ; intermediate code collected on this list
```

```
(define generate-icode
  (lambda (exp)
    (let* ((ignore (reset-label-counter))
           (startlabel (new-label "START")))
      (begin
        (lib-init)
        (:= icode-to-be-emitted (null))
        (:= library-icode (null))
        (:= library-count 0)
        (:= sym-label-env (mk-empty-env add-sym))

        ;; Initialize icode list with translation of a return of the
        ;; top-level expression along with a label for the start of
        ;; the program.
        (:= icode-list
            (list (labeldef->icode startlabel)
                  (return->icode (translate exp standard-c-t-env))))

        ;; Translating the top-level expression caused some code,
        ;; e.g., the bodies of lambda's and letrec's, to be put on
        ;; an auxilary list to be emitted latter.  Now's the time to
        ;; emit it.
        (emit-delayed-ic)))))



(define icode-to-be-emitted
  ;; Icode to be emitted later is put on this list in blocks.  A block
  ;; is a list of icode.  Each block is emitted in order, i.e., the
  ;; order of the list specifies control flow.  No order between blocks
  ;; is guaranteed.  Usually each block starts with a labeldef->icode so
  ;; that other blocks can refer to it.
  (ref (null)))

(define emit-ic-later
  (lambda (codelist)
    (:= icode-to-be-emitted (cons codelist (^ icode-to-be-emitted)))))

(define emit-delayed-ic
  (lambda ()
    (letrec ((flatten ; turns a list of lists into a list
              (lambda (iclist-list iclist)
                (if (null? iclist-list)
                    iclist
                    (flatten (cdr iclist-list)
                             (append (car iclist-list) iclist))))))
      (begin
        (:= icode-list (append (^ icode-list)
                               (flatten (^ icode-to-be-emitted) (null))))
        (:= icode-list (append (^ icode-list)
                               (flatten (^ library-icode) (null))))))))
```

```
(define library-icode
  ;; The library functions also generate delayed icode.  To make the output
  ;; clearer (i.e., to hide the ugly library stuff at the end of the outputed
  ;; code), we put the library icode on a separate list.  Before translating
  ;; library code, call swap-delayed-icode-lists to swap the
  ;; icode-to-be-emitted list with another list for library code.  When the
  ;; library code is done, the lists should be swapped back.
  (ref (null)))

(define library-count
  ;; When compiling library code, we may have to compile other library
  ;; procedures.  To make sure that the delayed icode list gets set
  ;; back to the user's list when we stop compiling library code, we
  ;; keep count of how deep we are into compiling libray calls within
  ;; library calls.
  (ref 0))

(define swap-delayed-icode-lists
  (lambda ()
    (let ((tmp (^ icode-to-be-emitted)))
      (begin (:= icode-to-be-emitted (^ library-icode))
             (:= library-icode tmp)))))

(define enter-library
  (lambda ()
    (begin (if (= 0 (^ library-count)) (swap-delayed-icode-lists) the-unit)
           (:= library-count (+ (^ library-count) 1)))))

(define leave-library
  (lambda ()
    (begin (:= library-count (- (^ library-count) 1))
           (if (= 0 (^ library-count)) (swap-delayed-icode-lists) the-unit))))
```

```
;;
;; Top-level dispatch for translation
;;
(define translate
  (lambda (exp c-t-env)
    (match exp
      ((int->exp^ _ n)                (trans-integer n))
      ((char->exp^ _ c)               (trans-integer (char->int c)))
      ((bool->exp^ _ b)               (trans-integer (if b 1 0)))
      ((string->exp^ _ s)             (trans-string s))
      ((sym->exp^ _ s)                (trans-symbol s))
      ((variable->exp^ tr var)        (trans-variable c-t-env (^ tr) var))
      ((conditional->exp^ tr t c a)   (trans-conditional c-t-env (^ tr) t c a))
      ((begin->exp^ tr exprs)         (trans-begin c-t-env (^ tr) exprs))
      ((combination->exp^ tr op args) (trans-call c-t-env (^ tr) op args))
      ((abstraction->exp^ tr args b)  (trans-lambda c-t-env (^ tr) args b))
      ((binder->exp^ tr defs b)       (trans-let c-t-env (^ tr) defs b))
      ((recursion->exp^ tr defs b)    (trans-letrec c-t-env (^ tr) defs b)))))


;; Integers
;;
(define trans-integer (lambda (n) (int->icode integer-type n)))



;; String
;;
(define trans-string
  ;; Since strings are vectors of characters, we need a way to translate
  ;; a string literal into a vector.  We do this by making a symbol out
  ;; of the string literal, and using the sym->string to build the char
  ;; vector at run-time.  (sym->string is in lib.fx)
  (lambda (s)
    (let ((s2s-type (parse-type '(-> (sym) string))))
      (translate (combination->exp      ;; combination
                  (ref string-type)     ;; this is its type
                  (variable->exp (ref s2s-type) 'sym->string)
                                        ;; this is the function being called
                  (list (sym->exp (ref symbol-type) (string->sym s))))
                                        ;; these are the arguments.
                 standard-c-t-env))))


;; (symbol Name)
;;
(define trans-symbol
  (lambda (s)
    (labelref->icode symbol-type ((^ sym-label-env) s))))
```

```
(define add-sym
  ;; If symbol not already in the environment, then make a label for
  ;; it and put the new label in the environ.  Look in table.fx to see
  ;; how mk-empty-env and mk-binder work.
  (lambda (key)
    (let ((label (new-label "SYM")))
      (begin
        (emit-ic-later (list (labeldef->icode label) (sym->icode key)))
        (:= sym-label-env ((mk-binder sym=?) key label (^ sym-label-env)))
        label))))

(define sym-label-env (ref (mk-empty-env add-sym)))



;; Identifiers
;;
(define trans-variable
  (lambda (c-t-env ty var)
    (match (c-t-lookup c-t-env var)
      ;;  compile a normal var ref as its access path...
      ((back+over->binding^ back over) (var->icode ty back over))

      ;; but compile prims as if they were (lambda (x y) (+ x y)).
      ((primitive->binding^ prim)
       (if (prim-constant? prim)
           (prim-constant-form prim)     ;; For space, newline, etc. constants
           (prim-closure-form prim)))))) ;; For primitive routines (e.g., +)


;; (if E1 E2 E3)
;;
(define trans-conditional
  (lambda (c-t-env ty tst con alt)
    (op->icode ty 'if (list (translate tst c-t-env)
                            (translate con c-t-env)
                            (translate alt c-t-env)))))


;; (begin E*)
;;
(define trans-begin
  (lambda (c-t-env ty exprs)
    (op->icode ty 'begin (map (lambda (x) (translate x c-t-env)) exprs))))
```

```
;; (lambda (I*) E)  -- create a procedure
;;
(define trans-lambda
  (lambda (c-t-env ty formals body)
    (let ((label (new-label "LAMBDA")))
      (begin
        ;; 1. Translate body but emit it later.
        (emit-ic-later (list (labeldef->icode label)
                             (body->icode
                              ty
                              (length formals)
                              (return->icode
                               (translate body (c-t-bind formals c-t-env))))))
        ;; 2. Translate the creation of the closure now
        (trans-closure ty label)))))

(define trans-closure
  ;;  The ENV reg is defined (as a reg number) in backend/gen.fx
  (lambda (ty proc-label)
    (alloc->icode ty 2 (list (labelref->icode unknown-type proc-label)
                             (reg->icode unknown-type ENV)))))


;; (let ((I E)*) E0)
;; For now, compile this as ((lambda I* E0) E*);
;; Can be fancier in the future
;;
(define trans-let
  (lambda (c-t-env ty defs body)
    (let ((names (map definition-name defs))
          (vals  (map definition-value defs)))
      (translate (combination->exp
                  (ref (expression-type body))
                  (abstraction->exp
                   (ref (make-arrow-type (map expression-type vals)
                                         (expression-type body)))
                   names
                   body)
                  vals)
                 c-t-env))))
```

```
;; (letrec ((I E)*) E0)
;;    The ENV reg is defined (as a reg number) in backend/gen.fx
;;
(define trans-letrec
  (lambda (c-t-env ty defs body)
    (let* ((label (new-label "LETREC"))
           (new-env (c-t-bind (map definition-name defs) c-t-env))
           (trans-vals (lambda (d) (translate d new-env))))
      (begin
        ;; 1. Translate the body of the letrec, but emit it later
        (emit-ic-later
         (list (labeldef->icode label)
               (letrec->icode
                (length defs)
                (map trans-vals (map definition-value defs))
                (return->icode (translate body new-env)))))
        ;; 2. Translate a call to the letrec body
        (op->icode ty
                   'call
                   (list (trans-closure (make-arrow-type '() ty) label)))))))


;; (E0 E*)
;;
(define trans-call
  (lambda (c-t-env ty op operands)
    (match op
      ((variable->exp^ _ var)
       (match (c-t-lookup c-t-env var)
         ((primitive->binding^ prim)
          ;;
          ;; Yes, all this was to pull out this one case.
          ;; Primitives get compiled in-line instead of as calls.
          ;;
          (trans-primitive-combination ty prim operands c-t-env))
         (_ (trans-unknown-combination ty op operands c-t-env))))
      (_ (trans-unknown-combination ty op operands c-t-env)))))

(define trans-primitive-combination
  ;; Trans a call to a primitive procedure
  (lambda (ty prim operands c-t-env)
    ((prim-inline-form prim)
     ty
     (map (lambda (a) (translate a c-t-env)) operands))))

(define trans-unknown-combination
  ;; Trans a call to an "unknown" (i.e. computed) procedure
  (lambda (ty operator operands c-t-env)
    (op->icode ty
               'call
               (cons (translate operator c-t-env)
                     (map (lambda (a) (translate a c-t-env)) operands)))))
```

```
;; new-label : symbol -> sexpr
;;
(define label-counter (ref 0))

(define reset-label-counter
  (lambda () (:= label-counter 0)))

(define new-label
  (lambda (prefix)
    (begin (:= label-counter (+ (^ label-counter) 1))
           (string-append (string-append prefix "_")
                          (int->string (^ label-counter))))))



;;
;;  The stuff that follows is all for displaying intermediate code
;;  (lists of labeled trees) prettily.
;;
(define unparse-icode
  (lambda (code)
    (letrec ((up-ic-list (lambda (cl) (map unparse-ic cl))))
      (match code
        ((labeldef->icode^ lab) lab)
        ((noop->icode^) 'nop)
        ((int->icode^ _ n) n)
        ((string->icode^ s) (list 'string-lit s))
        ((sym->icode^ s) (list 'symbol-lit s))
        ((word->icode^ w) (list 'word-lit w))
        ((labelref->icode^ _ lab) lab)
        ((reg->icode^ _ reg) (list 'r reg))
        ((op->icode^ _ s clist) (cons s (map unparse-icode clist)))
        ((alloc->icode^ _ i clist) (list 'alloc i (map unparse-icode clist)))
        ((var->icode^ _ back over) (list 'var back over))
        ((body->icode^ _ n b) `(body (,n) ,(unparse-icode b)))
        ((letrec->icode^ n l e)
         `(letrec ,(map unparse-icode l) ,(unparse-icode e)))
        ((return->icode^ code) (list 'return (unparse-icode code)))))))

(define display-icode
  (lambda (code)
    (match code
      ((labeldef->icode^ lab)
       (begin (newline) (display lab) (display ":")))
      (_ (begin (display "   ") (pp (unparse-icode code)))))))
```

```
(define old-display-icode      ;; for list-of-trees-style sc...
  (lambda (code)
    (letrec
        ((sp (lambda () (display " ")))
         (print-ic-list (lambda (cl)
                          (begin
                            (display "( ")
                            (for-each (lambda (e) (print-icode e) (sp)) cl)
                            (display ")"))))
         (print-icode
          (lambda (code)
            (match code
              ((labeldef->icode~ lab)          ; label definitions
               (begin (display lab) (display ":") ))
              ((noop->icode~)                  ; no-op; ignore it.
               the-unit)
              ((int->icode~ _ n)               ; immediate integers
               (begin (display n)))
              ((string->icode~ s)              ; immediate strings
               (begin (display ".string ") (display s) ))
              ((sym->icode~ s)                 ; immediate symbols
               (begin (display ".symbol ") (display s) ))
              ((word->icode~ w)
               (begin (display ".word ") (display w) ))
              ((labelref->icode~ _ lab)        ; label reference
               (begin (display lab)))
              ((reg->icode~ _ reg)             ; register reference
               (begin (display "r.") (display reg)))
              ((op->icode~ _ s clist)          ; operation
               (begin (display s) (sp) (print-ic-list clist)))
              ((alloc->icode~ _ i clist)       ; mem allocation
               (begin (display "alloc/") (display i) (print-ic-list clist)))
              ((var->icode~ _ back over)
               (begin (display "var(")
                      (display back) (display ",")
                      (display over) (display ")")))
              ((body->icode~ _ n b)
               (begin (display "body/") (display n)
                      (display " ") (print-icode b)))
              ((letrec->icode~ n l e)
               (begin (display "letrec/") (display n)
                      (display "(")
                      (for-each (lambda (x)
                                  (begin (print-icode x) (display " // ")))
                                  ;; the // terminates icode trees to make
                                  ;; them more easily readable.
                                l)
                      (display ") ")
                      (print-icode e)))
              ((return->icode~ code)
               (begin (display "return ") (print-icode code)))))))
      (match code
        ((labeldef->icode~ _) (print-icode code)) ; special hack for indenting
        (_ (begin (display "   ") (print-icode code)))))))
```

C.6 compiler/ic2oc.fx

The contents of the file compiler/ic2oc.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

;;; gen.fx -- translate list of trees to DLX ocode

;; The code generation is by recursive-decent tree rewriting.  The
;; intermediate code tree is traversed depth-first by various
;; routines.  When one of these routines recognizes the tree it is
;; looking at, it emits code for the tree and re-writes the tree into
;; another tree that indicates where the result was put.  For example,
;; a (op->icode t1 '+ (reg->icode _ 1) (reg->icode _ 2)) will be recognized
;; by the binary operation routine, which will emit the code
;; "add r3, r2, r1" and will rewrite the tree into (reg->icode t1 3).

;; The top level translation procedure is gencode.  Its function is to
;; translate root nodes into no-ops, and to translate intermediates
;; and leaves into either reg->icode nodes or int->icode nodes for
;; small integers.  Passing up the small integers allows us to take
;; advantage of operations w/ special forms for small immediates.  The
;; various routines called by gencode first translate the subtrees of
;; a node by calling gencode recursively, then translating the
;; simplified node through case analysis.  Thus we use
;; recursive-decent parsing to ''divide and conquer.''


(define ocode-list (ref (null)))        ; generated code will go here.

(define emit
  (lambda (op rands) (:= ocode-list (cons (insn op rands) (^ ocode-list)))))

(define emit-lw
  (lambda (dst slot block) ; reg#:int slot#:int reg#:int
    (emit 'lw (load->rands dst (otag slot) block))))

(define emit-sw
  (lambda (slot block src) ; reg#:int slot#:int reg#:int
    (emit 'sw (store->rands (otag slot) block src))))

(define emit-error      ; for better error reporting (and locating!)
  (lambda (s)
    (begin
      (display s)
      (newline)
      (emit 'error (err->rands s)))))
```

```
;; input comes from icode-list; output goes to ocode-list.
;;
(define generate-ocode
  (lambda ()
    (begin
      (:= ocode-list (null))
      (init-reg-allocator)
      (for-each (lambda (x) (gencode x)) (^ icode-list))
      (:= ocode-list (reverse-list (^ ocode-list))))))


;;
;; Top-level code generators -- translate any kind of icode node.
;;

(define gencode ; (-> (icode) icode)
  ;; Translate roots into noop->icode; all others into either
  ;; reg->icode or int->icode (for small ints).  In the register case,
  ;; gencode will allocate a register, and gencode's caller must
  ;; deallocate it.
  (lambda (code)
    (match code
     ((noop->icode^)                   code)
     ((labeldef->icode^ l)             (gen-labdef l))
     ((body->icode^ ty num-args body)  (gen-body ty num-args body))
     ((letrec->icode^ nvars vars body) (gen-letrec nvars vars body))
     ((return->icode^ e)               (gen-return e))
     ((string->icode^ s)               (gen-string s))
     ((sym->icode^ s)                  (gen-symbol s))
     ((word->icode^ w)                 (gen-word w))

     ((op->icode^ ty o l)              (gen-op ty o l))
     ((alloc->icode^ ty i codelist)    (gen-alloc ty i codelist))

     ((labelref->icode^ ty l)          (gen-labref ty l))
     ((int->icode^ ty n)               (gen-intref ty n))
     ((reg->icode^ _ r)                code)
     ((var->icode^ ty back over)       (gen-varref ty back over)))))

(define full-gencode ; (-> (icode) icode)
  ;; Same as gencode except no int->icode are returned
  (lambda (code)
    (let ((code2 (gencode code)))
      (match code2
        ((int->icode^ ty n)
         (let ((target (allocate-reg)))
           (begin
             (emit 'addi (rri->rands target ZERO (int->string (itag n))))
             (reg->icode ty target))))
        (_ code2)))))
```

```
(define target-gencode
  ;; Same as gencode except value is stored into specified register
  (lambda (code target)
    (let ((ty (match (gencode code)
                ((reg->icode~ ty r)
                 (if (= r target)
                     ty
                     (begin (emit 'or (rrr->rands target r ZERO))
                            (deallocate-reg r)
                            ty)))
                ((int->icode~ ty i)
                 (begin
                   (emit 'addi (rri->rands target ZERO (int->string (itag i))))
                   ty))
                (_ unit-type))))
      (reg->icode ty target))))



;;
;; Translation for roots of intermediate code
;;

(define gen-labdef
  (lambda (l) (begin (emit 'labeldef (label->rands l)) (noop->icode))))


(define gen-string
  (lambda (s) (begin (emit 'stringdef (string->rands s)) (noop->icode))))


(define gen-symbol
  (lambda (s) (begin (emit 'stringdef (string->rands (sym->string s)))
                     (noop->icode))))


(define gen-word
  (lambda (w) (begin (emit 'worddef (word->rands w))  noop->icode))))


(define gen-return      ; (-> (icode) icode)
  ;; Emit code to compute return result into VAL register, then emit
  ;; code to jump to caller (return address taken from activation
  ;; frame).
  (lambda (e)
    (begin
      (target-gencode e VAL)
      (emit-lw ATEMP 2 FP)
      (emit 'jr (r->rands ATEMP))
      (emit 'nop (nop->rands))
      (noop->icode))))
```

```
;; Generate code for a lambda body.  Assumptions: reg ARG0 contains
;; the address of the closure used to call us, and regs ARG1..ARGn
;; contain our params.  The closure is the pair (codeaddr environmentptr).

(define gen-body
  (lambda (ty num-args body)
    (let ((save-args (map (lambda (x) (reg->icode unknown-type x))
                          (integers-between ARG1 (+ ARG0 num-args))))
          (stack-env? (stack-allocate-environment-frame? (ictype body) body)))
      (begin
        ;; Allocate new env frame, link into static chain (available in
        ;; closure passed in ARG0), and save parameter values into it.
        (gen-alloc-block stack-env? (+ num-args 1) ENV) ; New frame
        (emit-lw ATEMP 1 ARG0)            ; Head of old static chain
        (emit-sw 0 ENV ATEMP)             ; link new frame into static chain
        (gen-fill-block ENV 1 save-args) ; Sv args in new env
        (gencode body)))))

(define gen-letrec
  ;; Same as lambda except we fill the new environment with evaluations
  ;; of the letrec variables instead of with argument registers.
  (lambda (num-vars vars body)
    (let ((stack-env? (stack-allocate-environment-frame? (ictype body) body)))
      (begin
        (gen-alloc-block stack-env? (+ num-vars 1) ENV) ; Alloc env frame
        (emit-lw ATEMP 1 ARG0)                          ; Lnk in2 static chain
        (emit-sw 0 ENV ATEMP)
        (gen-fill-block ENV 1 vars)     ; Fill frame slots w/ letrec variables
        (gencode body)))))

(define stack-allocate-environment-frame?
  (lambda (ty body) #f))


;;
;; Compile forms that call closures.
;;
(define gen-call
  ;; Remember that first argument in arglist is the closure to call.
  (lambda (ty arglist)
    (let ((label (new-label "RETURN"))
          (stack-act? (stack-allocate-activation-frame? ty arglist)))
      (begin
        (gen-begin-activation ; Emit code to allocate and fill activation frame
         stack-act? FrameSize label)
        (simulate-stackframe-push)     ; Since reg's are saved, we can use them
        (gen-arguments arglist)        ; Emit code to compute arguments
        (emit-lw ATEMP 0 ARG0)         ; Get address from closure
        (emit 'jr (r->rands ATEMP))
        (emit 'nop (nop->rands))
        (emit 'labeldef (label->rands label)) ; Here's the return label
        (gen-end-activation)           ; Emit code to restore registers
        (simulate-stackframe-pop)
```

```
        (let ((result (allocate-reg))) ; Alloc a reg to put result
          (begin (emit 'or (rrr->rands result VAL ZERO))
                 (reg->icode ty result)))))))

(define stack-allocate-activation-frame?
  (lambda (ty arglist) #f))

(define gen-jump
  ;; Generates a jump rather than a call to a closure.  The first
  ;; argument in arg list is the closure to jump to.
  (lambda (ty arglist)
    (begin
      (simulate-stackframe-push)
      (gen-arguments arglist)
      (emit-lw ATEMP 0 ARG0)
      (emit 'jr (r->rands ATEMP))
      (emit 'nop (nop->rands))
      (simulate-stackframe-pop)
      (int->icode ty 0))))

(define gen-arguments
  ;; compile args into arg regs.  return last-used arg reg.
  (lambda (arglist)
    (letrec ((loop (lambda (l r)
                     (if (null? l)
                         the-unit
                         (begin (allocate-specific-reg r)
                                (target-gencode (car l) r)
                                (loop (cdr l) (+ r 1)))))))
      (loop arglist ARG0))))



(define gen-intref ; (-> (int) icode)
  ;; Small integer values are are passed up.  For larger integers, a
  ;; register is allocated for the int and code is emitted to move the
  ;; int into the register.
  (lambda (ty n)
    (cond ;; ((= n 0) (reg->icode ty ZERO)) NO WRK <= way dst rgs R pckd
          ((int16? (itag n)) (int->icode ty n))
          (else (let ((target (allocate-reg))
                      (tn (int->string (itag n))))
                  (begin
                   (if (and (> n 0) (< n 32768))
                       (emit 'addui (rri->rands target ZERO tn))
                       (begin (emit 'lhi (ri->rands target (msw tn)))
                              (emit 'ori (rri->rands target target (lsw tn)))))
                   (reg->icode ty target)))))))
```

```
(define gen-labref ; (-> (string) icode)
  ;; Allocate reg and emit code to move value of label into it.
  (lambda (ty lab)
    (let ((target (allocate-reg)))
      (begin
        (emit 'lhi (ri->rands target (msw lab)))
        (emit 'ori (rri->rands target target (lsw lab)))
        (reg->icode ty target)))))


(define gen-varref
  (lambda (ty back over)
    (let ((target (allocate-reg))
      (begin (emit-lw target over (localref back))
             (reg->icode ty target)))))

(define localref ; (-> (int) int)
  ;; Emits code to walk up static chain.  Returns register pointing to
  ;; frame; local variable can be accessed by loading relative to this
  ;; pointer.
  (lambda (back)
    (letrec ((walk-up (lambda (n)
                        (if (= n 0)
                            the-unit
                            (begin
                              (emit-lw ATEMP 0 ATEMP)
                              (walk-up (- n 1)))))))
      (if (= back 0)
          ENV
          (begin (emit-lw ATEMP 0 ENV)
                 (walk-up (- back 1))
                 ATEMP)))))


;;
;; Translation for operators
;;
```

```
(define gen-if
  (lambda (ty args)
    (let ((treg (reg-num (full-gencode (car args))))
          (con (car (cdr args)))
          (alt (car (cdr (cdr args))))
          (elselabel (new-label "ELSE"))
          (joinlabel (new-label "JOIN")))
      (begin
        (emit 'beqz (ri->rands treg elselabel))
        (emit 'nop (nop->rands))
        (deallocate-reg treg)
          (target-gencode con VAL)
          (emit 'j (i->rands joinlabel))
          (emit 'nop (nop->rands))
        (emit 'labeldef (label->rands elselabel))
          (target-gencode alt VAL)
          ;; (emit 'j (i->rands joinlabel))     ; let this case fall through...
          ;; (emit 'nop (nop->rands))
        (emit 'labeldef (label->rands joinlabel))
        (let ((result (allocate-reg)))         ; Alloc a reg for result
          (begin (emit 'or (rrr->rands result VAL ZERO))
                 (reg->icode ty result)))))))


(define gen-begin
  ;; Gen code for a list of expressions, returning value of last in
  ;; list.  Incoming list must have at least one element.
  (lambda (ty codelist)
    (let ((result (gencode (car codelist))))
      (if (null? (cdr codelist))
          result
          (begin
            (match result ((reg->icode^ _ r) (deallocate-reg r)) (_ the-unit))
            (gen-begin ty (cdr codelist)))))))


(define gen-alloc
  ;; Generate code to allocate and fill a block of memory whose size
  ;; is fixed at compile time.  Used for allocating cons cells, pairs,
  ;; and closures.
  (lambda (ty size codelist)
    (let ((blkptr (allocate-reg)))
      (begin
        (gen-alloc-block #f size blkptr)      ; Gen call to blk alloc'er
        (gen-fill-block blkptr 0 codelist) ; Fill blk w/ values
        (reg->icode ty blkptr)))))
```

```
(define gen-alloc-block
  ;; Emit code to call allocation routine, putting address of
  ;; allocated block in register whose number is passed in blkptr.
  (lambda (stack-alloc? size blkptr)
    (begin
      (emit 'ori (rri->rands ATEMP ZERO (int->string (itag size))))
      (if stack-alloc?
          (emit 'jal (i->rands "_SALLOC"))
          (emit 'jal (i->rands "_ALLOC")))
      (emit 'nop (nop->rands))
      (emit 'or (rrr->rands blkptr ZERO ATEMP)))))

(define gen-fill-block
  ;; Emit code to fill a block with values.  The values to fill the
  ;; block with are the results of evaluating codelist.  The slots to
  ;; put these results start at offset(blkptr) and go up from there.
  ;; "offset" is slot number.
  (lambda (blkptr offset codelist)
    (if (null? codelist)
        the-unit
        (let ((src (reg-num (full-gencode (car codelist)))))
          (begin
            (emit-sw offset blkptr src)
            (deallocate-reg src)
            (gen-fill-block blkptr (+ offset 1) (cdr codelist)))))))


(define gen-begin-activation
  ;; Emit code to allocate and fill an activation frame.  Updates FP
  ;; register (and SP if stack-allocate? is true).  Trashes ATEMP and
  ;; RETADR.
  (lambda (stack-allocate? Fsize return-label)
    (gen-alloc-block stack-allocate? Fsize ATEMP)
    (emit-sw 0 ATEMP FP) ; Save old frame pointer
    (emit 'or (rrr->rands FP ZERO ATEMP)) ; Load new frame into FP
    (if stack-allocate?
        (begin
          (emit 'addi (rri->rands ATEMP SP (int->string (* 4 (+ Fsize 1)))))
          (emit-sw 1 FP ATEMP)) ; Save old value of stack pointer
        (emit-sw 1 FP SP))
    (emit 'lhi (ri->rands ATEMP (msw return-label)))
    (emit 'ori (rri->rands ATEMP ATEMP (lsw return-label)))
    (emit-sw 2 FP ATEMP) ; Save return address
    (emit-sw 3 FP ENV)   ; Save environment pointer
    (emit 'jal (i->rands "_SAVE"))
    (emit 'nop (nop->rands))))
```

```
(define gen-end-activation
  ;; Emit code to restore registers saved in activation frame.
  ;; Trashes ATEMP and RETADR.
  (lambda ()
    (emit 'jal (i->rands "_RESTORE"))
    (emit 'nop (nop->rands))
    (emit-lw ENV 3 FP)
    (emit-lw SP 1 FP)
    (emit-lw FP 0 FP)))


(define gen-not
  (lambda (ty args)
    (let ((reg (reg-num (full-gencode (car args)))))
      (begin
        (emit 'xori (rri->rands reg reg (int->string (itag 1))))
        (reg->icode ty reg)))))


(define gen-assign ; (-> ((listof icode)) icode)
  (lambda (ty args)
    (let ((sreg (reg-num (full-gencode (cadr args))))
          (dreg (reg-num (full-gencode (car args)))))
      (begin
        (emit-sw 0 dreg sreg)
        (deallocate-reg dreg)
        (deallocate-reg sreg)
        (int->icode ty (itag 0))))))
```

```
(define gen-load ; (-> ((listof icode)) icode)
  ;; The first operand is a base pointer, i.e., a tagged pointer to a
  ;; block of memory created by alloc.  The second is an integer index
  ;; pointing to a slot inside that block of memory.  Those slots are
  ;; numbered consecutively 0, 1, 2, ...  The index must be turned
  ;; into a byte offset, which is 4x the index number, then added into
  ;; the base, either with an explicit add or with the
  ;; load-with-displacement ocode.
  (lambda (ty args)
    (let ((base (reg-num (full-gencode (car args))))
          (disp (gencode (cadr args))))
      (match disp
        ((int->icode^ _ i)
         (begin
           (if (int16? (otag i))
               (emit-lw base i base)
               (let ((dispr (reg-num (full-gencode disp))))
                 (emit 'slli (rri->rands dispr dispr "1"))
                 (emit 'add (rrr->rands base base dispr))
                 (emit-lw base 0 base)
                 (deallocate-reg dispr)))
           (reg->icode ty base)))
        (_ (let ((dispr (reg-num (full-gencode disp))))
             (emit 'slli (rri->rands dispr dispr "1"))
             (emit 'add (rrr->rands base base dispr))
             (emit-lw base 0 base)
             (deallocate-reg dispr)
             (reg->icode ty base)))))))
```

```
(define gen-vec-alloc
  ;; Generate code to allocate and fill a block of memory where both the
  ;; size and the contents of the block are computed at compile time.  All
  ;; slots of the block are filled with the same value.  Used for allocating
  ;; vectors.
  (lambda (ty codelist)
    (let* ((blkptr (allocate-reg))
           (size (reg-num (full-gencode (car codelist))))
           (filler (reg-num (full-gencode (cadr codelist))))
           (loop (new-label "LOOP"))
           (test (new-label "TEST")))
      (begin
          (emit 'or (rrr->rands ATEMP ZERO size))
          (emit 'jal (i->rands "_ALLOC"))
          (emit 'nop (nop->rands))
          (emit 'or (rrr->rands blkptr ZERO ATEMP))
          (emit 'j (i->rands test))
          (emit 'nop (nop->rands))
        (emit 'labeldef (label->rands loop))
          (emit-sw 0 ATEMP filler)
          (emit 'addi (rri->rands ATEMP ATEMP (int->string 4)))
        (emit 'labeldef (label->rands test))
          (emit 'subi (rri->rands size size (int->string 2)))
          (emit 'bnez (ri->rands size loop))
          (emit 'nop (nop->rands))

        (deallocate-reg size)
        (deallocate-reg filler)
        (reg->icode ty blkptr)))))


(define gen-vec-length
  (lambda (ty codelist)
    (let ((vec (reg-num (full-gencode (car codelist)))))
      (begin
        (emit-lw vec -1 vec)
        (reg->icode ty vec)))))
```

```
(define gen-vec-set!
  (lambda (ty codelist)
    (let* ((vec (reg-num (full-gencode (car codelist))))
           (idx (gencode (cadr codelist)))
           (val (reg-num (full-gencode (cadr (cdr codelist))))))
      (begin
        (match idx
          ((int->icode^ _ i)
           (if (int16? (otag i))
               (emit-sw i vec val)
               (let ((dispr (reg-num (full-gencode idx))))
                 (begin
                   (emit 'slli (rri->rands dispr dispr "1"))
                   (emit 'add (rrr->rands vec vec dispr))
                   (emit-sw 0 vec val)
                   (deallocate-reg dispr)))))
          ((reg->icode^ _ dispr)
           (begin
             (emit 'slli (rri->rands dispr dispr "1"))
             (emit 'add (rrr->rands vec vec dispr))
             (emit-sw 0 vec val)
             (deallocate-reg dispr))))
        (deallocate-reg val)
        (deallocate-reg vec)
        (int->icode ty 0)))))


(define gen-sym2string
  (lambda (ty codelist)
    (let ((sym (reg-num (full-gencode (car codelist)))))
      (begin
        (emit 'or (rrr->rands ATEMP sym ZERO))
        (emit 'jal (i->rands "_SYM2STRING"))
        (emit 'nop (nop->rands))
        (emit 'or (rrr->rands sym ATEMP ZERO))
        (reg->icode ty sym)))))


(define gen-put-char
  (lambda (ty codelist)
    (begin
      (target-gencode (car codelist) ATEMP)
      (emit 'jal (i->rands "_TPUTCHAR")) ; arg is tagged, so use tagged vers.
      (emit 'nop (nop->rands))
      (int->icode ty 0))))
```

```
(define gen-op  ; (-> (op (listof icode)) icode)
  ;; The code generators for each type of operator are kept in a table
  ;; indexed by the operation symbol.  As mentioned before, code
  ;; generation is by recursive decent: we translate sub-trees first by
  ;; calling gencode, then translate the operator.
  (lambda (ty op args) ((op-table op) ty args)))


(define make-binop-gen
  ;; Code generators for alu binary op's all look the same, so abstract.
  (lambda (op opi)
    (lambda (ty args)
      (let ((l (reg-num (full-gencode (car args))))
            (r (gencode (cadr args))))
        (begin
          (match r
           ((int->icode~ _ n)
            (emit opi (rri->rands l l (int->string (itag n)))))
           ((reg->icode~ _ rreg)
            ;; Do the general thing
            (begin (emit op (rrr->rands l l rreg))
                   (deallocate-reg rreg))))
          (reg->icode ty l))))))


(define make-binop-gen2
  ;; Binary operators w/ no built-in ALU operation
  (lambda (emitter)
    (lambda (ty args)
      (let ((l (reg-num (full-gencode (car args))))
            (r (reg-num (full-gencode (cadr args)))))
        (begin (emitter l r)
               (deallocate-reg r)
               (reg->icode ty l))))))

(define emit-mul
  (lambda (l r)
    (begin (emit 'movi2fp (rr->rands FP0 l))
           (emit 'srai (rri->rands ATEMP r "1"))
           (emit 'movi2fp (rr->rands FP1 ATEMP))
           (emit 'mult (rrr->rands FP2 FP0 FP1))
           (emit 'movfp2i (rr->rands l FP2)))))

(define emit-div
  (lambda (l r)
    (begin (emit 'movi2fp (rr->rands FP0 l))
           (emit 'movi2fp (rr->rands FP1 r))
           (emit 'div (rrr->rands FP2 FP0 FP1))
           (emit 'movfp2i (rr->rands l FP2))
           (emit 'slli (rri->rands l l "1")))))
```

```
(define emit-remainder
  (lambda (l r)
    (begin (emit 'movi2fp (rr->rands FP0 l))
           (emit 'movi2fp (rr->rands FP1 r))
           (emit 'div (rrr->rands FP2 FP0 FP1))
           (emit 'mult (rrr->rands FP3 FP2 FP1))
           (emit 'movfp2i (rr->rands ATEMP FP3))
           (emit 'sub (rrr->rands l l ATEMP)))))


(define op-table
  (let ((empty (lambda (bad-op)
                 (emit-error (string-append "bad op: " (sym->string key)))))
        (op-list
         '((+ ,(make-binop-gen 'add 'addi))
           (- ,(make-binop-gen 'sub 'subi))
           (seq ,(make-binop-gen 'seq 'seqi))
           (slt ,(make-binop-gen 'slt 'slti))
           (sle ,(make-binop-gen 'sle 'slei))
           (sgt ,(make-binop-gen 'sgt 'sgti))
           (sge ,(make-binop-gen 'sge 'sgei))
           (* ,(make-binop-gen2 emit-mul))
           (/ ,(make-binop-gen2 emit-div))
           (remainder ,(make-binop-gen2 emit-remainder))
           (:= ,gen-assign)
           (^ ,gen-load)
           (vec-alloc ,gen-vec-alloc)
           (vec-set ,gen-vec-set!)
           (vec-length ,gen-vec-length)
           (sym2string ,gen-sym2string)
           (put-char ,gen-put-char)
           (not ,gen-not)
           (call ,gen-call)
           (jump ,gen-jump)
           (if ,gen-if)
           (begin ,gen-begin))))
    (list2env empty sym=? op-list)))


;; Tags:
;;   The least significant bit of each machine word is used as a tag so that
;;   the garbage collector can distinguish pointers from integers.  All
;;   integer values are shifted up one bit position (i.e., multiplied by
;;   two) to have a zero tag.  Pointers (which all point to things on word
;;   boundaries, and so are all multiples of four) are given a tag of 1.  To
;;   dereference a pointer, one must subtract one from it before indirecting
;;   through it.
;;
(define itag (lambda (i) (* 2 i)))                    ; tag an integer
(define de-itag (lambda (i) (quotient i 2)))
```

```
(define otag
  ;; For referencing into object slots.  Takes a slot number and
  ;; returns an offset which, when added to a pointer, addresses the
  ;; desired data value.  (Remember slot -1, the size of the object,
  ;; is stored at address "tagged_pointer - 1," slot 0 at
  ;; "tagged_pointer + 3," etc.)
  (lambda (slot) (+ (* slot 4) 3)))

(define de-otag (lambda (i) (/ (- i 3) 4)))

(define int16? ; (-> (int) bool)
  (lambda (n) (and (<= n 32767) (>= n -32768))))

(define msw
  (lambda (l)
    (string-append "((" (string-append l ")>>16)&0x0000ffff"))))

(define lsw
  (lambda (l) (string-append "(" (string-append l ")&0xffff"))))



(define FrameSize 28)

;; Symbolic names for some registers (local to this file with one
;; exception noted below).

(define ZERO    0)
(define VAL     1)
(define ENV     2)       ; this def is used in exp2ic.fx
(define FP      3)
(define SP      4)
(define HP      5)
(define ARG0    6)
(define ARG1    7)
(define ARG2    8)
(define ARG3    9)
(define ARG4    10)
(define ARG5    11)
(define ARG6    12)
(define ARG7    13)
(define ARG8    14)
(define ARG9    15)
(define TOPARG  29)
(define ATEMP   30)
(define RETADR  31)

(define FP0 0)
(define FP1 1)
(define FP2 2)
(define FP3 3)
(define FP4 4)
(define FP5 5)
```

```
;;
;; Register allocation
;;

;; The following def's are local to the allocator

(define *free-regs* (ref (generate-vector 0 (lambda (i) #t))))
(define *free-reg-stack* (ref '()))
(define *num-allocatable* (+ 1 (- TOPARG ARGO)))

(define allocatable-reg? (lambda (r) (and (<= ARGO r) (<= r TOPARG))))
(define reg-free?
  (lambda (r) (and (allocatable-reg? r)
                   (vector-ref (^ *free-regs*) (- r ARGO)))))

(define take-reg! (lambda (r) (vector-set! (^ *free-regs*) (- r ARGO) #f)))
(define put-reg! (lambda (r) (vector-set! (^ *free-regs*) (- r ARGO) #t)))

;; The following def's are exported by the allocator, but are local to
;; this file.

(define init-reg-allocator
  (lambda ()
    (begin
      (:= *free-reg-stack* '())
      (:= *free-regs* (generate-vector *num-allocatable* (lambda (i) #t))))))

(define allocate-reg
  (lambda ()
    (letrec ((loop (lambda (r) (cond ((> r TOPARG) (error "Out of registers."))
                                     ((reg-free? r) (begin (take-reg! r) r))
                                     (else (loop (+ r 1)))))))
      (loop ARGO))))

(define allocate-specific-reg
  (lambda (r)
    (if (reg-free? r)
        (begin (take-reg! r) r)
        (error "Register needed twice."))))

(define deallocate-reg
  (lambda (reg)
    (if (allocatable-reg? reg)
        (put-reg! reg)
        the-unit)))

(define highest-used-reg
  (lambda ()
    (letrec ((loop (lambda (r) (cond ((< r ARGO) ARGO)
                                     ((not (reg-free? r)) r)
                                     (else (loop (- r 1)))))))
      (loop TOPARG))))
```

```
(define simulate-stackframe-push
  (lambda ()
    (begin
      (:= *free-reg-stack* (cons (^ *free-regs*) (^ *free-reg-stack*)))
      (:= *free-regs* (generate-vector *num-allocatable* (lambda (i) #t))))))

(define simulate-stackframe-pop
  (lambda ()
    (begin
      (:= *free-regs* (car (^ *free-reg-stack*)))
      (:= *free-reg-stack* (cdr (^ *free-reg-stack*))))))
```

C.7 compiler/icode.fx

The contents of the file compiler/icode.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

(define-datatype icode
  ;; tree roots:
  ;;
  (labeldef->icode string)                   ; label definitions
  (body->icode type int icode)               ; stack-act stack-env
  (letrec->icode int (listof icode) icode)   ; num-vars vars body
  (noop->icode)                              ; no-op
  (string->icode string)                     ; immediate strings
  (sym->icode sym)                           ; immediate symbols

  ;; intermediate nodes
  ;;
  (return->icode icode)                             ; return an expression
  (op->icode type sym (listof icode))        ; primitive operations
  (alloc->icode type int (listof icode))     ; mem allocation

  ;; leaf nodes
  ;;
  (labelref->icode type string)              ; refs to labels
  (int->icode type int)                      ; immediate integers
  (reg->icode type int)                      ; register ref (for codegen...)
  (var->icode type int int)                  ; var refs (back, over)
  (word->icode string))                      ; For labels used as ints.  This
                                             ; node can also be a root!

(define noop (noop->icode)) ; only need one of these...

(define reg-num ; quick access to register numbers
  (lambda (icode) (match icode ((reg->icode^ _ n) n))))
```

```
(define ictype
  ;; Find the high-level type associated with a piece of icode. Note that
  ;; this operation is not appropriate for all icodes. For example, there
  ;; is no type associated with labeldef->icode. The only root with a
  ;; type slot is the body slot: this type does not give the type of the
  ;; expression, but rather is just a way to communicate down the types
  ;; of the arguments. return->icode does not have a type slot because
  ;; the type of that expression is just the type of the value being returned.
  ;; word->icode does not have a type slot because it is used for internal
  ;; compiler stuff.
  (lambda (ic)
    (match ic
      ((body->icode^ ty _ _) ty)
      ((letrec->icode^ _ _ ic) (ictype ic))
      ((return->icode^ ic) (ictype ic))
      ((op->icode^ ty _ _) ty)
      ((alloc->icode^ ty _ _) ty)
      ((labelref->icode^ ty _) ty)
      ((int->icode^ ty _) ty)
      ((reg->icode^ ty _) ty)
      ((var->icode^ ty _ _) ty))))
```

C.8 compiler/lib.fx

The contents of the file compiler/lib.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

;; lib.fx -- compile-time environments and built-ins

;; This file contains the compile-time environments and built-in
;; primitives for mico-FX built-ins. There are two compile time
;; environments, one for the type reconstructor (standard-type-env)
;; and another for the exp2ic translator (standard-c-t-env). This
;; file exports the following variables:

;; lib-init:(-> () unit)
;;     Initializes stuff internal to library; needs to be called once
;;   per compilation.

;; c-t-lookup:(-> (c-t-env sym) binding)
;;     Returns the binding for sym; aborts with error if not found.
;; c-t-bind:(-> ((listof sym) c-t-env) c-t-env)
;;     Returns new environment with a new lexical level containing
;;   the list of symbols pushed onto the old environment.
;; standard-c-t-env:c-t-env
;;     Standard environment defined by microFX (containing bindings
;;   for +, -, etc.).
;;
;;   The c-t-env abstract type is defined by operations in this file. The
;;   environment itself is an abstract type defined by two operations:
```

```
;; (define-datatype binding
;;   (back+over->binding int int)
;;   (primitive->binding prim))
;;
;;   The "binding" data type is returned by c-t-lookup.  back+over is
;;   for variables defined by the code being compiled: back indicates
;;   how far down the static chain the variable resides (0 ==>
;;   variable is in current environment), over indicates how far over
;;   in the environment frame the variable resides.
;;
;;   "prim" is an abstract data type used for variables defined by
;;   microFX.  It has the operations:
;;
;; prim-constant?:(-> (prim) bool)
;;     Returns true iff primitive is a constant (eg, the-unit)
;;   rather than a function (eg, +).
;; prim-inline-form:(-> (prim) (-> ((listof icode)) icode))
;;     For functional primitives, returns a function that takes a
;;   list of icode which are the arguments to the function and
;;   returns icode that computes the function on those arguments.
;; prim-closure-form:(-> (prim) icode)
;;     For functional primitives, returns icode which is a closure
;;   for the indicated primitive.
;;
;;   Note that prim-inline-form and prim-closure-form can modify the
;;   library-icode list, a list of auxilary icode needed by library
;;   functions.

(define lib-init
  (lambda () (:= prim-closure-env (mk-empty-env add-prim-closure))))

(define-datatype prim
  ;; The following data structure is used inside this module to hold
  ;; important data about the built-ins in the library.  For built-ins
  ;; coded in assembly language, the structure holds the label of the
  ;; routine.  For library routines coded in micro-FX, the structure
  ;; contains the unparsed micro-FX expresssion for the routine.  For
  ;; library routines that can be inlined (e.g., +, - *), the
  ;; structure holds both an unparsed expresssion for the routine plus
  ;; a procedure that takes a list of arguments (in icode form) and
  ;; returns icode for the inlined procedure.  The first is used in
  ;; situations like '+, while the second is used in situations like (+ x y).
  (asm->prim ty label)          ; prim coded in assy lang (in ~microFX/runtime)
  (cnst->prim icode)            ; prim is a constant
  (lib->prim sexp)              ; prim coded in microFX
  (inline->prim sexp (-> ((listof icode)) icode)))      ; inline-able prim
```

```
(define-datatype binding
  ;; Bindings held in the compile-time environment.  Primitives are
  ;; wrapped up as uniqueof's so we can have a table of them (using eq?).
  ;; (The type (uniqueof t) denotes the set of values for which each
  ;; element of type is distinguishable.  In other words, it's a promise
  ;; to the compiler that there's no sharing of elements, so eq? will
  ;; works.  The current minifx interpreter doesn't actually do anything
  ;; with this information, but there's always tomorrow...)
  ;;
  (primitive->binding (uniqueof prim))
  (back+over->binding int int))


;; Code implementing c-t-env used by exp2ic.  We don't use our
;; usual environments here so we can encode back+over in the
;; lambda's that make up the environment.
(define empty-c-t-env (lambda (var) (error "unbound variable"  var)))

(define c-t-lookup (lambda (c-t-env var) (c-t-env var)))

(define c-t-add-prims
  (lambda (ops prims env)
    (letrec ((loop (lambda (var ops prims)
                     (if (null? ops)
                         (env var)
                         (if (same-variable? (car ops) var)
                             (primitive->binding (car prims))
                             (loop var (cdr ops) (cdr prims)))))))
      (lambda (var) (loop var ops prims)))))

(define c-t-bind
  (lambda (formals c-t-env)
    (lambda (var)
      (letrec ((loop (lambda (i formals)
                       (if (null? formals)
                           (match (c-t-lookup c-t-env var)
                             ((back+over->binding~ back over)
                              (back+over->binding (+ back 1) over))
                             (binding binding))
                           (if (same-variable? var (car formals))
                               (back+over->binding 0 i)
                               (loop (+ i 1) (cdr formals)))))))
        (loop 1 formals)))))

(define prim-constant?
  (lambda (prim)
    (match (value prim)
      ((cnst->prim~ ic) #t)
      (_ #f))))

(define prim-constant-form ;; (-> (prim) icode)
  (lambda (prim)
    (match (value prim)
      ((cnst->prim~ ic) ic))))
```

```
(define prim-inline-form ;; (-> (prim) (-> (ty (listof icode)) icode))
  (lambda (prim)
    (match (value prim)
      ((inline->prim^ _ emitter) emitter)
      (_ (lambda (ty args)
           (op->icode ty 'call (cons (prim-closure-form prim) args)))))))

(define prim-closure-form ;; (-> (prim) icode)
  (lambda (prim) ((^ prim-closure-env) prim)))

;; Closures for library routines are created only once and are
;; memoized in prim-closure-env, defined below.

(define add-prim-closure
  ;; If primitive not already in the environment, then make a closure
  ;; icode for it and insert that icode into the table
  (lambda (prim)
    (begin (enter-library) ;; Trans lib fn's w/ separate delay list
           (let ((closure (make-prim-closure prim)))
             (begin (:= prim-closure-env
                        ((mk-binder eq?) prim closure (^ prim-closure-env)))
                    (leave-library) ;; Back to normal
                    closure)))))

(define prim-closure-env (ref (mk-empty-env add-prim-closure)))

(define make-prim-closure
  ;; There's no closure for this prim yet (first-time reference).  Make one.
  (lambda (prim)
    (match (value prim)
      ((asm->prim^ ty label)           ; asm prim: grab closure.
       (trans-closure ty label))
      ((inline->prim^ sexp _)          ; inline prim: create and compile one.
       (let* ((exp (parse sexp))
              (type (reconstruct-top exp)))
         (translate exp standard-c-t-env)))
      ((lib->prim^ sexp)               ; same for lib prims.
       (let* ((exp (parse sexp))
              (type (reconstruct-top exp)))
         (translate exp standard-c-t-env))))))

;; Functions to create icode for inlined primitive operations

(define mk-op-prim ;; + - * quotient ...
  (lambda (op) (lambda (ty arglist) (op->icode ty op arglist))))

(define mk-cons-prim ;; Data constructor: cons, pair, ref
  (lambda (size)
    (lambda (ty arglist) (alloc->icode ty size arglist))))
```

```
(define mk-sel-prim
  ;; Projection functions: car, cdr, left, right, etc.
  ;; Tagging implements a check for (car (null)) or (cdr (null)) !
  (lambda (offset)
    (lambda (ty arglist)
      (op->icode ty '^ (list (car arglist)
                             (int->icode integer-type offset))))))

(define assign-prim ;; For :=
  (lambda (ty arglist) (op->icode ty ':= arglist)))

(define neg-prim
  (lambda (ty arglist) (op->icode ty 'sub (cons (int->icode ty 0) arglist))))

(define null-prim
  ;; The empty list is represented as immediate 0.  It's untagged so
  ;; the garbage collector doesn't try to follow it.
  (lambda (ty arglist) (int->icode ty 0)))

(define null?-prim
  (lambda (ty arglist)
    (op->icode ty 'seq (list (car arglist) (int->icode integer-type 0)))))

(define sym->string-prim
  (lambda (ty arglist) (op->icode ty 'sym2string arglist)))

(define cvt-prim ;; Pass argument straight thru -- for type coersion
  (lambda (ty arglist) (car arglist)))

(define put-char-prim
  (lambda (ty arglist) (op->icode ty 'put-char arglist)))

(define void-prim
  ;; Cons up a fake argument.  In real life, this routine should generate
  ;; code to crash the system.
  (lambda (ty arglist) (int->icode ty 0)))

(define void-name
  ;; Make it hard for the user to get at the void function
  (string->symbol "the void function"))

(define no-prim
  (lambda (ty arglist) (error "Unimplemented primitive used.")))

;; Most built in functions are written in micro-FX code.  This code is
;; listed below.  These expressions are evaluated in the standard
;; compile time environment, built-ins can call one another.

;; Char prims
(define mk-char-ci-pred
  (lambda (op) `(lambda (x y) (,op (char-downcase x) (char-downcase y)))))

(define l-char-alphabetic?
  '(lambda (x) (or (char-lower-case? x) (char-upper-case? x))))
```

```
(define l-char-numeric?
  '(lambda (x)
     (and (>= (char->int x) (char->int #\0))
          (<= (char->int x) (char->int #\0)))))

(define l-char-whitespace?
  '(lambda (x)
     (or (= (char->int x) (char->int space))
         (= (char->int x) (char->int tab))
         (= (char->int x) (char->int page))
         (= (char->int x) (char->int newline)))))

(define l-char-lower-case?
  '(lambda (x)
     (and (>= (char->int x) (char->int #\a))
          (<= (char->int x) (char->int #\z)))))

(define l-char-upper-case?
  '(lambda (x)
     (and (>= (char->int x) (char->int #\A))
          (<= (char->int x) (char->int #\Z)))))

(define l-char-upcase
  '(lambda (x)
     (if (char-lower-case? x)
         (int->char (+ (char->int #\A) (- (char->int x) (char->int #\a))))
         x)))

(define l-char-downcase
  '(lambda (x)
     (if (char-upper-case? x)
         (int->char (+ (char->int #\a) (- (char->int x) (char->int #\A))))
         x)))

;; List prims

;; (define l-set-car!
;; (define l-set-cdr!

(define l-length
  '(lambda (l)
     (letrec ((loop (lambda (lst len)
                      (if (null? lst) len (loop (cdr lst) (+ len 1))))))
       (loop l 0))))

;; (define l-append

(define l-reverse        ; minifx doesn't have this built-in.
  '(lambda (l)
     (letrec ((loop (lambda (lst rev)
                      (if (null? lst)
                          rev
                          (loop (cdr lst) (cons (car lst) rev))))))
       (loop l (null)))))
```

```
;; (define l-list-tail
;; (define l-list-ref
;; (define l-map
;; (define l-for-each
;; (define l-reduce
(define l-list->string '(lambda (l) (vector->string (list->vector l))))
(define l-string->list '(lambda (s) (vector->list (string->vector s))))

;; Strings
;; (define l-string-fill!
;; (define l-string=?
;; (define l-string<?
;; (define l-string>?
;; (define l-string<=?
;; (define l-string>=?
;; (define l-string-ci=?
;; (define l-string-ci<?
;; (define l-string-ci>?
;; (define l-string-ci<=?
;; (define l-string-ci>=?
;; (define l-substring
(define l-string-append
  '(lambda (s1 s2)
     (letrec ((r (make-string (+ (string-length s1) (string-length s2)) #\X))
              (loop (lambda (s i j)
                      (if (>= i (string-length s))
                          the-unit
                          (begin
                            (string-set! r j (string-ref s i))
                            (loop s (+ i 1) (+ j 1)))))))
       (begin (loop s1 0 0) (loop s2 0 (string-length s1)) r))))

;; (define l-string-copy

;; Syms
;; (define l-sym->string
;; (define l-string->sym
;; (define l-sym=?
;; (define l-hash

;; Vectors
(define l-make-vector (mk-op-prim 'vec-alloc))
(define l-vector-length (mk-op-prim 'vec-length))
(define l-vector-ref (mk-op-prim '^))
(define l-vector-set! (mk-op-prim 'vec-set))

;; (define l-vector-fill!

(define l-vector->list
 '(letrec ((v2l (lambda (v i l)
                  (if (= i -1)
                      l
                      (v2l v (- i 1) (cons (vector-ref v i) l))))))
    (lambda (v) (v2l v (- (vector-length v) 1) (null)))))
```

```
(define l-list->vector
  '(letrec ((l2v (lambda (l i v)
                   (if (= i (vector-length v))
                       v
                       (begin (vector-set! v i (car l))
                              (l2v (cdr l) (+ i 1) v))))))
     (lambda (l)
       (if (= (length l) 0)
           (make-vector (length l) (,void-name))
           (l2v l 0 (make-vector (length l) (car l)))))))

;; (define l-vector-map
;; (define l-vector-map2
;; (define l-vector-reduce
;; (define l-scan
;; (define l-segmented-scan
;; (define l-compress
;; (define l-expand
;; (define l-eoshift

;; Unparsers
(define l-unparse-bool '(lambda (x) (if x "#t" "#f")))
(define l-unparse-char '(lambda (x) (string-append "#\\" (make-string 1 x))))
(define l-unparse-unit '(lambda (x) "#u"))

(define l-unparse-int
  '(lambda (x)
    (letrec ((loop (lambda (l i)
                     (if (= i 0)
                         (vector->string (list->vector l))
                         (loop (cons (int->char
                                       (+ (remainder i 10) (char->int #\0)))
                                     l)
                               (/ i 10))))))
      (if (= x 0)
          "0"
          (if (< x 0)
              (string-append "-" (loop (null) x))
              (loop (null) x))))))


(define l-unparse-string
  '(lambda (x) (string-append "\"" (string-append x "\""))))

(define l-unparse-symbol
  '(lambda (x) (string-append "(symbol " (string-append (sym->string x) ")"))))
```

```
(define l-unparse-list
  '(lambda (pr x)
    (letrec ((loop
              (lambda (s x)
                (if (null? x)
                    (string-append s ")")
                    (loop (string-append s (string-append " " (pr (car x))))
                          (cdr x))))))
      (loop "(list" x))))

(define l-unparse-vector
  '(lambda (pr x)
    (string-append "(list->vector "
                   (string-append (unparse-list pr (vector->list x)) ")"))))

(define l-unparse-pair
  '(lambda (prl prr x)
    (string-append
     "(pair "
     (string-append (prl (left x)) (string-append (prr (right x)) ")")))))

(define l-put-string
  '(lambda (s)
     (letrec ((loop
               (lambda (i)
                 (if (>= i (string-length s))
                     the-unit
                     (begin (put-char (string-ref s i)) (loop (+ i 1)))))))
       (loop 0))))

;; The list "standard-prim-bindings" contains typing and other
;; information about micro-fx built-ins.  This list is used to
;; initialize two other lists, the standard type environment used by
;; the type checker and the standard compile-time environment used by
;; the expression to icode translator.

;; Each entry of the list has the form "(op type prim)" where op is
;; the name of the built-in (a symbol), type is the type of the
;; primitive (an sexp), and prim is the translation information for
;; the built-in (a "prim" sum-of-products).

;; The following routines make it easy to define entries in
;; standard-prim-bindings:

(define in0
  (lambda (op type icode-emitter)
    (list op type
          (unique (inline->prim '(lambda () (,op)) icode-emitter)))))

(define in1
  (lambda (op type icode-emitter)
    (list op type
          (unique (inline->prim '(lambda (x) (,op x)) icode-emitter)))))
```

```
(define in2
  (lambda (op type icode-emitter)
    (list op type
          (unique (inline->prim `(lambda (x y) (,op x y)) icode-emitter)))))

(define in3
  (lambda (op type icode-emitter)
    (list op type
         (unique (inline->prim `(lambda (x y z) (,op x y z)) icode-emitter)))))

(define cnst
  (lambda (op type icode)
    (list op type (unique (cnst->prim icode)))))

(define lib
  (lambda (op type sexp)
    (list op type (unique (lib->prim sexp)))))

(define asm
  (lambda (op type label)
    (list op
          type
          (unique (asm->prim (instantiate-schema (parse-schema type))
                             label))))))

(define standard-bindings
`(
 ,(cnst 'the-unit 'unit (int->icode unit-type 0))
 ,(asm 'printf '(generic (t) (-> (sym t) unit)) "PRINTF")
 ,(in1 'put-char '(-> (char) unit) put-char-prim)
 ,(lib 'put-string '(-> (string) unit) l-put-string)
 ,(in0 void-name '(generic (t) (-> () t)) void-prim)

 ;; Booleans
 ,(in2 'equiv? '(-> (bool bool) bool) (mk-op-prim 'seq))
 ,(in2 'and? '(-> (bool bool) bool) (mk-op-prim 'and))
 ,(in2 'or? '(-> (bool bool) bool) (mk-op-prim 'or))
 ,(in1 'not? '(-> (bool) bool) (mk-op-prim 'not))
 ,(in1 'not '(-> (bool) bool) (mk-op-prim 'not))
```

```
;; Characters
,(cnst 'backspace 'char (int->icode character-type 8))
,(cnst 'newline 'char (int->icode character-type 10))
,(cnst 'page 'char (int->icode character-type 12))
,(cnst 'space 'char (int->icode character-type 32))
,(cnst 'tab 'char (int->icode character-type 9))
,(in2 'char=? '(-> (char char) bool) (mk-op-prim 'seq))
,(in2 'char<? '(-> (char char) bool) (mk-op-prim 'slt))
,(in2 'char>? '(-> (char char) bool) (mk-op-prim 'sgt))
,(in2 'char<=? '(-> (char char) bool) (mk-op-prim 'sle))
,(in2 'char>=? '(-> (char char) bool) (mk-op-prim 'sge))
,(lib 'char-ci=? '(-> (char char) bool) (mk-char-ci-pred char=?))
,(lib 'char-ci<? '(-> (char char) bool) (mk-char-ci-pred char<?))
,(lib 'char-ci>? '(-> (char char) bool) (mk-char-ci-pred char>?))
,(lib 'char-ci<=? '(-> (char char) bool) (mk-char-ci-pred char<=?))
,(lib 'char-ci>=? '(-> (char char) bool) (mk-char-ci-pred char>=?))
,(lib 'char-alphabetic? '(-> (char) bool) l-char-alphabetic?)
,(lib 'char-numeric? '(-> (char) bool) l-char-numeric?)
,(lib 'char-whitespace? '(-> (char) bool) l-char-whitespace?)
,(lib 'char-lower-case? '(-> (char) bool) l-char-lower-case?)
,(lib 'char-upper-case? '(-> (char) bool) l-char-upper-case?)
,(lib 'char-upcase '(-> (char) char) l-char-upcase)
,(lib 'char-downcase '(-> (char) char) l-char-downcase)
,(in1 'char->int '(-> (char) int) cvt-prim)
,(in1 'int->char '(-> (int) char) cvt-prim)

;; Integers
,(in2 '= '(-> (int int) bool) (mk-op-prim 'seq))
,(in2 '< '(-> (int int) bool) (mk-op-prim 'slt))
,(in2 '> '(-> (int int) bool) (mk-op-prim 'sgt))
,(in2 '<= '(-> (int int) bool) (mk-op-prim 'sle))
,(in2 '>= '(-> (int int) bool) (mk-op-prim 'sge))
,(in2 '+ '(-> (int int) int) (mk-op-prim '+))
,(in2 '- '(-> (int int) int) (mk-op-prim '-))
,(in2 '* '(-> (int int) int) (mk-op-prim '*))
,(in2 '/ '(-> (int int) int) (mk-op-prim '/))
,(in2 'quotient '(-> (int int) int) (mk-op-prim '/))   ; helps in testing...
,(in1 'neg '(-> (int) int) neg-prim)
,(in2 'remainder '(-> (int int) int) (mk-op-prim 'remainder))
,(in2 'modulo '(-> (int int) int) no-prim)
,(lib 'abs '(-> (int) int) '(lambda (x) (if (< x 0) (- 0 x) x)))
```

```
;; Lists
,(in1 'null? '(generic (t) (-> ((listof t)) bool)) null?-prim)
,(in0 'null '(generic (t) (-> () (listof t))) null-prim)
,(in2 'cons '(generic (t) (-> (t (listof t)) (listof t))) (mk-cons-prim 2))
,(in1 'car '(generic (t) (-> ((listof t)) t)) (mk-sel-prim 0))
,(in1 'cdr '(generic (t) (-> ((listof t)) (listof t))) (mk-sel-prim 1))
,(in2 'set-car! '(generic (t) (-> ((listof t) t) unit)) no-prim)
,(in2 'set-cdr! '(generic (t) (-> ((listof t) (listof t)) unit)) no-prim)
,(lib 'length '(generic (t) (-> ((listof t)) int)) l-length)
,(in2 'append '(generic (t) (-> ((listof t) (listof t)) (listof t))) no-prim)
,(in1 'reverse '(generic (t) (-> ((listof t)) (listof t))) no-prim)
,(in2 'list-tail '(generic (t) (-> ((listof t) int) (listof t))) no-prim)
,(in2 'list-ref '(generic (t) (-> ((listof t) int) t)) no-prim)
,(in3 'map
      '(generic (t1 t2) (-> ((-> (t1) t2) (listof t1)) (listof t2)))
      no-prim)
,(in3 'for-each
      '(generic (t1 t2) (-> ((-> (t1) t2) (listof t1)) unit))
      no-prim)
,(in3 'reduce
      '(generic (t1 t2) (-> ((-> (t1) t2) (listof t1) t2) t2))
      no-prim)
,(lib 'list->string '(-> ((listof char)) string) l-list->string)
,(lib 'string->list '(-> (string) (listof char)) l-string->list)

;; Ordered pairs
,(in2 'pair '(generic (t1 t2) (-> (t1 t2) (pairof t1 t2))) (mk-cons-prim 2))
,(in1 'left '(generic (t1 t2) (-> ((pairof t1 t2)) t1)) (mk-sel-prim 0))
,(in1 'right '(generic (t1 t2) (-> ((pairof t1 t2)) t2)) (mk-sel-prim 1))

;; Refs
,(in1 'ref '(generic (t) (-> (t) (refof t))) (mk-cons-prim 2))
,(in1 '^ '(generic (t) (-> ((refof t)) t)) (mk-sel-prim 0))
,(in2 ':= '(generic (t) (-> ((refof t) t) unit)) assign-prim)
```

```
;; Strings
,(in2 'make-string '(-> (int char) string) l-make-vector)
,(in2 'string-length '(-> (string) int) l-vector-length)
,(in2 'string-ref '(-> (string int) char) l-vector-ref)
,(in2 'string-set! '(-> (string int char) unit) l-vector-set!)
,(in2 'string-fill! '(-> (string char) unit) no-prim)
,(in2 'string=? '(-> (string string) bool) no-prim)
,(in2 'string<? '(-> (string string) bool) no-prim)
,(in2 'string>? '(-> (string string) bool) no-prim)
,(in2 'string<=? '(-> (string string) bool) no-prim)
,(in2 'string>=? '(-> (string string) bool) no-prim)
,(in2 'string-ci=? '(-> (string string) bool) no-prim)
,(in2 'string-ci<? '(-> (string string) bool) no-prim)
,(in2 'string-ci>? '(-> (string string) bool) no-prim)
,(in2 'string-ci<=? '(-> (string string) bool) no-prim)
,(in2 'string-ci>=? '(-> (string string) bool) no-prim)
,(in3 'substring '(-> (string int int) string) no-prim)
,(lib 'string-append '(-> (string string) string) l-string-append)
,(in1 'string-copy '(-> (string) string) no-prim)
,(in1 'string->vector '(-> (string) (vectorof char)) cvt-prim)
,(in1 'vector->string '(-> ((vectorof char)) string) cvt-prim)

;; Syms
,(in1 'sym->string '(-> (sym) string) sym->string-prim)
,(in1 'string->sym '(-> (string) sym) no-prim)
,(in2 'sym=? '(-> (sym sym) bool) (mk-op-prim 'seq))
,(in1 'hash '(-> (sym) int) no-prim)
```

```
;; Vectors
,(in2 'make-vector '(generic (t) (-> (int t) (vectorof t))) l-make-vector)
,(in1 'vector-length
      '(generic (t) (-> ((vectorof t)) int))
      l-vector-length)
,(in2 'vector-ref
      '(generic (t) (-> ((vectorof t) int) t))
      l-vector-ref)
,(in3 'vector-set!
      '(generic (t) (-> ((vectorof t) int t) unit))
      l-vector-set!)
,(in2 'vector-fill! '(generic (t) (-> ((vectorof t) t) unit)) no-prim)
,(lib 'vector->list
      '(generic (t) (-> ((vectorof t)) (listof t)))
      l-vector->list)
,(lib 'list->vector
      '(generic (t) (-> ((listof t)) (vectorof t)))
      l-list->vector)
,(in2 'vector-map
      '(generic (t1 t2) (-> ((-> (t1) t2) (vectorof t1)) (vectorof t2)))
      no-prim)
,(in3 'vector-map2
      '(generic (t1 t2 t3)
         (-> ((-> (t1 t2) t3) (vectorof t1) (vectorof t2)) (vectorof t3)))
      no-prim)
,(in3 'vector-reduce
      '(generic (t1 t2) (-> ((-> (t1 t2) t2) (vectorof t1) t2) t2))
      no-prim)
,(in2 'scan
      '(generic (t) (-> ((-> (t t) t) (vectorof t)) (vectorof t)))
      no-prim)
,(in3 'segmented-scan
      '(generic (t)
         (-> ((-> (t t) t) (vectorof bool) (vectorof t)) (vectorof t)))
      no-prim)
,(in2 'compress
      '(generic (t1) (-> ((vectorof bool) (vectorof t)) (vectorof t)))
      no-prim)
,(in3 'expand
      '(generic (t1)
         (-> ((vectorof bool) (vectorof t) (vectorof t)) (vectorof t)))
      no-prim)
,(in3 'eoshift
      '(generic (t1) (-> (int (vectorof t) (vectorof t)) (vectorof t)))
      no-prim)
```

```
;; Unparsers
,(lib 'unparse-bool '(-> (bool) string) l-unparse-bool)
,(lib 'unparse-char '(-> (char) string) l-unparse-char)
,(lib 'unparse-int '(-> (int) string) l-unparse-int)
,(lib 'unparse-string '(-> (string) string) l-unparse-string)
,(lib 'unparse-symbol '(-> (sym) string) l-unparse-symbol)
,(lib 'unparse-unit '(-> (unit) string) l-unparse-unit)
,(lib 'unparse-list
      '(generic (t) (-> ((-> (t) string) (listof t)) string))
      l-unparse-list)
,(lib 'unparse-vector
      '(generic (t) (-> ((-> (t) string) (vectorof t)) string))
      l-unparse-vector)
,(lib 'unparse-pair
      '(generic (r l)
         (-> ((-> (l) string) (-> (r) string) (pairof r l)) string))
      l-unparse-pair)
))

;; Define standard-type-environment and standard-c-t-env.

(define sb-name (lambda (binding) (car binding)))
(define sb-type (lambda (binding) (parse-schema (cadr binding))))
(define sb-prim (lambda (binding) (cadr (cdr binding))))

(define standard-type-environment
  (extend-by-schemas empty-type-environment
                     (map sb-name standard-bindings)
                     (map sb-type standard-bindings)))

(define standard-c-t-env
  (c-t-add-prims (map sb-name standard-bindings)
                 (map sb-prim standard-bindings)
                 empty-c-t-env))
```

C.9 compiler/misc.fx

The contents of the file compiler/misc.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

(define id (lambda (x) x))

(define max (lambda (n1 n2) (if (> n1 n2) n1 n2)))

;; List routines
(define integers-between
  (lambda (lo hi)
    (if (> lo hi) (null) (cons lo (integers-between (+ 1 lo) hi)))))
```

```
(define reverse-list    ; minifx doesn't have this built-in.
  (lambda (lst)
    (letrec ((rlist (lambda (l r)
                      (if (null? l)
                          r
                          (rlist (cdr l) (cons (car l) r))))))
      (rlist lst (null)))))

(define reduce-left
  (lambda (fn lst seed)
    (letrec ((loop (lambda (l v)
                     (if (null? l)
                         v
                         (loop (cdr l) (fn v (car l)))))))
      (loop lst seed))))

(define for-each
  (lambda (proc l)
    (if (null? l)
        the-unit
        (begin (proc (car l))
               (for-each proc (cdr l))))))

(define for-each-2
  (lambda (proc lst1 lst2)
    (if (null? lst1)
        the-unit
        (begin
          (proc (car lst1) (car lst2))
          (for-each-2 proc (cdr lst1) (cdr lst2))))))

(define 1st car)
(define 2nd cadr)
(define 3rd caddr)
(define 4th cadddr)


;; Vector stuff


;; String stuff

(define map-string
  (lambda (proc str)
    (letrec ((len (string-length str))
             (loop (lambda (i)
                     (if (< i len)
                         (begin (string-set! str i (proc (string-ref str i)))
                                (loop (+ i 1)))
                         str))))
      (loop 0))))
```

```
(define char->string
  (lambda (c)
    (let ((s "x"))
      (begin (string-set! s 0 c)
             (string-copy s)))))

(define down-string
  (lambda (s) (map-string char-downcase s)))

(define up-string
  (lambda (s) (map-string char-upcase s)))

(define pad
  (lambda (s sz) (substring "      " 0 (- sz (string-length s)))))

(define down-sym
  (lambda (s) (down-string (symbol->string s))))

(define up-sym
  (lambda (s) (up-string (symbol->string s))))


;; Stream stuff

(define copy-input-stream-to-output-stream
  (let ((terminators (char-set #\newline)))
    (lambda (fin fout)
      (let ((line (read-string terminators fin)))
        (if (eof-object? line)
          the-unit
          (begin
            (*scheme-read-char* fin) ; clear the newline
            (display line fout) (newline fout)
            (copy-input-stream-to-output-stream fin fout)))))))


;; Wrappers for uniqueof functions (may not be the right semantics for
;; all data types...)
(define unique (lambda (x) x))
(define value (lambda (x) x))


;; Vector stuff

(define generate-vector
  (lambda (size proc)
    (letrec ((ans  (make-vector size))
             (loop
              (lambda (i)
                (if (= i size)
                    ans
                    (begin (vector-set! ans i (proc i))
                           (loop (+ i 1)))))))
      (loop 0))))
```

```
;; Faked-up implementation of tuples:

(define tuple list)
(define tuple-ref list-ref)
```

C.10 compiler/oc2txt.fx

The contents of the file compiler/oc2txt.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

;; oc2txt.fx --  output ocode to a file in official DLX assembly format

(define print-one-instruction
  (lambda (x strm)
    (begin
      (match x
       ((ocode~ 'labeldef (label->rands~ lab))
        (display (string-append lab ":") strm))
       (_ (display (string-append "      " (unparse-ocode x)) strm)))
      (newline strm))))

;;
;; Routines for printing stuff, including unparsing results from micro-FX
;; program run in DLX simulator
;;

(define extract-value
  ;; Unparse result from DLX simulator
  (lambda (word type)
    (match (prune type)
      ((base->type~ 'bool)
       (bool->sexp (not (= word 0))))
      ((base->type~ 'char)
       (char->sexp (int->char (quotient word 2))))
      ((base->type~ 'int)
       (int->sexp (quotient word 2)))
      ((base->type~ 'string)
       (string->sexp (extract-string word)))
      ((base->type~ 'sym)
       (sym->sexp (extract-symbol word)))
      ((base->type~ 'unit)
       (sym->sexp 'the-unit))
```

```
      ((compound->type~ '-> _)
       (unparse-type type))
      ((compound->type~ 'pairof et)
       (pair->sexp (cons (extract-value (get-slot word 0) (car et))
                         (extract-value (get-slot word 1) (cadr et)))))
      ((compound->type~ 'listof et)
       (list->sexp (extract-list word (car et))))
      ((compound->type~ 'vectorof et)
       (vector->sexp (extract-vec word (car et))))
      (_ '(unrecognized type ,(unparse-type type) ,word)))))

(define extract-string
  (lambda (word)
    (let* ((vc (extract-vec word (parse-type 'char))))
      (list->string (vector->list vc)))))

(define extract-symbol
  (lambda (word) (string->sym (get-mem word))))

(define extract-list
  (lambda (word type)
    (if (= word 0)
        (null)
        (cons (extract-value (get-slot word 0) type)
              (extract-list (get-slot word 1) type)))))

(define extract-vec
  (lambda (word type)
    (letrec ((len (de-itag (get-slot word -1)))
             (loop
              (lambda (v i)
                (if (= i len)
                    v
                    (begin
                      (vector-set! v i (extract-value (get-slot word i) type))
                      (loop v (+ i 1)))))))
      (loop (make-vector len) 0))))


(define type-to-printf-format
  ;; A munged version of unparse-type (../frontend/parse.fx) that prints
  ;; out a printf-like format string for printing the result of the
  ;; computation.  The printf-like code is written in DLX assembly code
  ;; in the runtime directory.
  (lambda (type)
    (if (recognize-type? type)
        (string-append "%" (type2printf type))
        (string-append "unrecognized type"))))
```

```
(define recognize-type?
  (lambda (type)
    (match (prune type)
      ((base->type^ t) (if (memq t '(int bool char string sym unit)) #t #f))
      ((compound->type^ 'listof _) #t)
      ((compound->type^ t sub-types)
       (and (memq t '(listof pairof vectorof refof ->))
            (reduce-left and? (map recognize-type? sub-types) #t)))
      ((unknown->type^) #f))))

(define type2printf
  (lambda (type)
    (match (prune type)
      ((base->type^ 'int) "d")
      ((base->type^ 'bool) "b")
      ((base->type^ 'char) "c")
      ((base->type^ 'string) "vc") ; strings are vec's of chars
      ((base->type^ 'sym) "s")
      ((base->type^ 'unit) "u")
      ((compound->type^ 'listof q)
       (match (prune (car q))
         ((tvariable->type^ _) "ld")
         (q1 (string-append "l" (type2printf q1)))))
      ((compound->type^ 'pairof operands)
       (string-append
        "p"
        (string-append (type2printf (car operands))
                       (type2printf (cadr operands)))))
      ((compound->type^ 'vectorof operands)
       (string-append "v" (type2printf (car operands))))
      ((compound->type^ 'refof operands)
       (string-append "r" (type2printf (car operands))))
      ((compound->type^ '-> operands) "F"))))
```

C.11 compiler/ocode.fx

The contents of the file compiler/ocode.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

;; object code format

(define insn cons)
(define op-code car)
(define op-rands cdr)
(define ocode^ cons^)
```

```
(define-datatype rands
  (rrr->rands int int int)
  (rri->rands int int string)
  (rr->rands int int)
  (ri->rands int string)
  (r->rands int)
  (i->rands string)
  (load->rands int int int)
  (store->rands int int int)
  (nop->rands)

  (symbol->rands sym)
  (string->rands string)
  (label->rands string)
  (word->rands string)

  (err->rands string))
```

C.12 compiler/optimize.fx

The contents of the file compiler/optimize.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

;; icode optimizer pass


(define optimize-icode
  (lambda ()
    (:= icode-list (map prop-returns-down (^ icode-list)))))



;; Tail call handling
;;
;; A tail call is any call that is immediately followed by a return (in
;; execution order).  We find them by pushing returns downward in the
;; tree, looking for (return (call foo)) and changing that to (jump foo).
;;
;; To push returns downward in a tree, we use the transformations:
;;
;;     (return (if a b c))           --> (if a (return b) (return c))
;;     (return (begin e1 e2 ... en)) --> (begin e1 e2 ... (return en))
;;
(define prop-returns-down
  (lambda (ic)
    (match ic
      ((body->icode^ ty n b) (body->icode ty n (prop-returns-down b)))

      ((letrec->icode^ n args b)
       (letrec->icode n args (prop-returns-down b)))
```

```
((return->icode~ (op->icode~ ty 'call args))
 (op->icode ty 'jump args))

((return->icode~ (op->icode~ ty 'if tac))
 (op->icode ty
            'if
            (list (car tac)
                  (prop-returns-down (return->icode (cadr tac)))
                  (prop-returns-down (return->icode (caddr tac))))))

((return->icode~ (op->icode~ ty 'begin exprs))
 (letrec ((p-r-down-last
           (lambda (l)
             (if (null? (cdr l))
                 (list (prop-returns-down (return->icode (car l))))
                 (cons (car l) (p-r-down-last (cdr l)))))))
   (op->icode ty 'begin (p-r-down-last exprs))))

(_ ic))))
```

C.13 compiler/parse.fx

The contents of the file compiler/parse.fx:

```
;;; -*- Mode: Scheme; Package: SCHEME -*-
; Expression and type parsers

;; Top-level parser

(define parse
  (lambda (sexpr) (parse-exp sexpr)))

;; Parse a single expression
(define parse-exp                               ; sexpr -> exp
  (lambda (sexpr)
    (match sexpr
      ((sym->sexp~ sym) (variable->exp (ref unknown-type) sym))

      ((char->sexp~ c) (char->exp (ref character-type) c))

      ((bool->sexp~ b) (bool->exp (ref boolean-type) b))

      ((int->sexp~ n) (int->exp (ref integer-type) n))

      ((string->sexp~ s) (string->exp (ref string-type) s))

      ;; First thing in list is a SYMBOL
      (`(,(sym->sexp~ head) ,@_) ((get-parser-for-keyword head) sexpr))

      ;; Procedure call is the default
      (`(,operator ,@operands) (parse-combination operator operands))

      (_ (error "unrecognized expression" sexpr)))))
```

```
;; Parse a definition
(define parse-definition
  (lambda (sexpr)
    (match sexpr
      ('(define ,name ,value)
       (make-definition (parse-formal name) (parse-exp value)))
      (_ (error "invalid definition" sexpr)))))

;; check-out a formal parameter; make sure it's not a reserved word.
(define parse-formal
  (lambda (sexpr)
    (match sexpr
      ((sym->sexp^ name)
       (if (memq name (^ all-keywords))
           (error "attempt to use reserved word as variable name"
                  sexpr)
           name))
      (_ (error "invalid variable name" sexpr)))))

;; Most of the rest of this file concerns itself with special forms
;; (expressions of the form (reserved-word ...)).  Define-keyword is a
;; function that defines a reserved word, associating it with a function
;; that can parse the named construct.

;; List of parsing functions.  Each checks to see if the new keyword
;; is its own, and either parses the whole thing or passes the buck.
(define keyword-table
  (ref (lambda (head)
         (lambda (sexpr)          ;; Procedure call is the default
           (match sexpr
             ('(,operator ,@operands) (parse-combination operator
             operands)) (_ (error "this shouldn't happen")))))))

(define all-keywords (ref (null))) ; list of keywords.

(define get-parser-for-keyword
  (lambda (name) ((^ keyword-table) name)))

(define define-keyword
  (lambda (keyword parser)
    (let ((current-table (^ keyword-table)))
      (begin (:= keyword-table
                 (lambda (head)
                   (if (eq? head keyword) parser (current-table head))))
             (:= all-keywords (cons keyword (^ all-keywords)))
             keyword))))

;;  And here are the parsing functions...
```

```
;; (symbol Name)
;;
(define-keyword 'symbol
  (lambda (sexpr)
    (match sexpr
      (`(symbol ,(sym->sexp^ name)) (sym->exp (ref symbol-type) name))
      (_ (parse-error sexpr)))))


;; (call E0 E*)
;;
(define-keyword 'call
  (lambda (sexpr)
    (match sexpr
      (`(call ,operator ,@operands)
       (parse-combination operator operands))
      (_ (parse-error sexpr)))))


;; (E0 E*)
;;
(define parse-combination
  (lambda (operator operands)
    (combination->exp (ref unknown-type)
                      (parse-exp operator)
                      (map parse-exp operands))))


;; (if E1 E2 E3)
;;
(define-keyword 'if
  (lambda (sexpr)
    (match sexpr
      (`(if ,test ,con ,alt)
       (conditional->exp (ref unknown-type)
                         (parse-exp test)
                         (parse-exp con)
                         (parse-exp alt)))
      (_ (parse-error sexpr)))))


;; (begin E1 ... En)
;;
(define-keyword 'begin
  (lambda (sexpr)
    (match sexpr
      (`(begin)        (parse-exp '(null)))
      (`(begin ,exp)   (parse-exp exp))
      (`(begin ,@exps) (begin->exp (ref unknown-type) (map parse-exp exps)))
      (_ (parse-error sexpr)))))
```

```
;; (lambda (I*) E)
;;
(define-keyword 'lambda
  (lambda (sexpr)
    (match sexpr
      (`(lambda (,@formals) ,body)
       (abstraction->exp (ref unknown-type)
                         (map parse-formal formals)
                         (parse-exp body)))
      (_ (parse-error sexpr)))))


;; LET is not simply sugar because handled specially during typechecking
;; (let ((I E)*) E0)
(define-keyword 'let
  (lambda (sexpr)
    (letrec ((parse-binding-spec
              (lambda (bspec)
                (match bspec
                  (`(,name ,value)
                   (make-definition (parse-formal name) (parse-exp value)))
                  (_ (error "invalid binding specifier" bspec))))))
      (match sexpr
        (`(let (,(sym->sexp^ _) ,_)) (parse-error sexpr)) ; LET without a body
        (`(let (,@bspecs) ,body)
         (binder->exp (ref unknown-type)
                      (map parse-binding-spec bspecs)
                      (parse-exp body)))
        (_ (parse-error sexpr))))))


;; (letrec ((I E)*) E0)
;;
(define-keyword 'letrec
  (lambda (sexpr)
    (letrec ((parse-binding-spec
              (lambda (bspec)
                (match bspec
                  (`(,name ,value)
                   (make-definition (parse-formal name) (parse-exp value)))
                  (_ (error "invalid binding specifier" bspec))))))
      (match sexpr
        (`(letrec (,(sym->sexp^ _) ,_)) (parse-error sexpr)) ; LETREC w/no body
        (`(letrec (,@bspecs) ,body)
         (recursion->exp (ref unknown-type)
                         (map parse-binding-spec bspecs)
                         (parse-exp body)))
        (_ (parse-error sexpr))))))

;; Sugars
```

```
;; (and)          ==>  #t
;; (and E)        ==>  E
;; (and E0 E+)    ==>  (if E0 (and E+) #f)
;;
(define-keyword 'and
  (lambda (sexpr)
    (match sexpr
      (`(and ,@exp-list)
       (parse-exp (letrec ((recur (lambda (exps)
                                    (match exps
                                      ((null~)           '#t)
                                      (`(,exp)            exp)
                                      ((cons~ first rest)
                                       `(if ,first ,(recur rest) #f))))))
                    (recur exp-list))))
      (_ (parse-error sexpr)))))


;; (or)          ==>  #f
;; (or E)        ==>  E
;; (or E0 E+)    ==>  (if E0 #t (or E+))
;;
(define-keyword 'or
  (lambda (sexpr)
    (match sexpr
      (`(or ,@exp-list)
       (parse-exp (letrec ((recur (lambda (exps)
                                    (match exps
                                     ((null~)            '#f)
                                     (`(,exp)            exp)
                                     ((cons~ first rest)
                                      `(if ,first #t ,(recur rest)))))))
                    (recur exp-list))))
      (_ (parse-error sexpr)))))


;; (list E*)
;;
(define-keyword 'list
  (lambda (sexpr)
    (match sexpr
      (`(list ,@exp-list)
       (parse-exp (letrec ((recur (lambda (exps)
                                    (match exps
                                      ((null~) '(null))
                                      ((cons~ first rest)
                                       `(cons ,first ,(recur rest)))))))
                    (recur exp-list))))
      (_ (parse-error sexpr)))))

(define parse-error
  (lambda (sexpr)
    (error "invalid expression syntax" sexpr)))
```

```
;; Unparser
;;
(define unparse
  (lambda (exp)
    (unparse-exp exp)))

(define unparse-exp
  (letrec ((unparse-binding-specs
            (lambda (defs)
              (map (lambda (def)
                     `(,(definition-name def)
                       ,(unparse-exp (definition-value def))))
                   defs))))
    (lambda (exp)
      (match exp
        ((variable->exp^ _ var) (sym->sexp var))
        ((bool->exp^ _ b) (bool->sexp b))
        ((int->exp^ _ n) (int->sexp n))
        ((string->exp^ _ s) (string->sexp s))
        ((char->exp^ _ c) (char->sexp c))
        ((sym->exp^ _ name) `(symbol ,(sym->sexp name)))
        ((conditional->exp^ _ test consequent alternate)
         `(if ,(unparse-exp test)
              ,(unparse-exp consequent)
              ,(unparse-exp alternate)))
        ((begin->exp^ _ exprs)
         `(begin ,@(map unparse-exp exprs)))
        ((abstraction->exp^ _ formals body)
         `(lambda (,@formals) ,(unparse-exp body)))
        ((combination->exp^ _ operator operands)
         `(,(unparse-exp operator)
           ,@(map unparse-exp operands)))
        ((binder->exp^ _ defs body)
         `(let (,@(unparse-binding-specs defs))
            ,(unparse-exp body)))
        ((recursion->exp^ _ defs body)
         `(letrec (,@(unparse-binding-specs defs))
            ,(unparse-exp body)))))))
```

```
;; Generic pretty printer.
;;   pprint-type is a function on types to include the reconstructed
;;   type information in the output.
;;
(define pprint-exp-gen
  (letrec ((pp-binding-specs
            (lambda (defs pprint-type)
              (map (lambda (def)
                     `(,(definition-name def)
                       ,(pprint-exp-gen (definition-value def) pprint-type)))
                   defs))))
    (lambda (exp pprint-type)
      (match exp
        ((variable->exp~ ty var)
         (list 'variable->exp (pprint-type ty) (sym->sexp var)))
        ((bool->exp~ ty b)
         (list 'bool->exp (pprint-type ty) (bool->sexp b)))
        ((int->exp~ ty n)
         (list 'int->exp (pprint-type ty) (int->sexp n)))
        ((char->exp~ ty c)
         (list 'char->exp (pprint-type ty) (char->sexp c)))
        ((string->exp~ ty s)
         (list 'string->exp (pprint-type ty) (string->sexp s)))
        ((sym->exp~ ty name)
         (list 'sym->exp (pprint-type ty) (sym->sexp name)))
        ((conditional->exp~ ty test consequent alternate)
         (list 'conditional->exp (pprint-type ty)
           (pprint-exp-gen test pprint-type)
           (pprint-exp-gen consequent pprint-type)
           (pprint-exp-gen alternate pprint-type)))
        ((begin->exp~ ty exprs)
         (cons 'begin->exp (cons (pprint-type ty)
                                 (map (lambda (e) (pprint-exp-gen e pprint-type))
                                      exprs))))
        ((abstraction->exp~ ty formals body)
         (list 'abstraction->exp (pprint-type ty)
               `(,@formals) (pprint-exp-gen body pprint-type)))
        ((combination->exp~ ty operator operands)
         (list 'combination->exp (pprint-type ty)
               (pprint-exp-gen operator pprint-type)
               (map (lambda (op) (pprint-exp-gen op pprint-type)) operands)))
        ((binder->exp~ ty defs body)
         (list 'binder->exp (pprint-type ty)
               (pp-binding-specs defs pprint-type)
               (pprint-exp-gen body pprint-type)))
        ((recursion->exp~ ty defs body)
         (list 'recursion->exp (pprint-type ty)
               (pp-binding-specs defs pprint-type)
               (pprint-exp-gen body pprint-type)))))))

;; pprinter.
;;
(define (pprint-exp exp) (pprint-exp-gen exp (lambda (ty) 'ty)))
```

```
;; pprint with types.
;;
(define (pprint-exp-types exp)
  (pprint-exp-gen exp (lambda (tref) (unparse-type (^ tref)))))

;; Type expression parser
;;
(define parse-type
  (lambda (sexpr)
    (match sexpr
      ((sym->sexp^ sym) (base->type sym))
      (`(-> (,@arg-types) ,result-type)
       (compound->type arrow-constructor
                       (cons (parse-type result-type)
                             (map parse-type arg-types))))
      (`(,(sym->sexp^ name) ,@types)
       (compound->type name (map parse-type types)))
      (_ (error "invalid type expression syntax" sexpr)))))

;; Type expression unparser
;;
(define unparse-type
  (lambda (type)
    (match (prune type)
      ((base->type^ sym) (sym->sexp sym))
      ((compound->type^ constructor operands)
       (if (same-constructor? constructor arrow-constructor)
           `(-> (,@(map unparse-type (cdr operands)))
                ,(unparse-type (car operands)))
           `(,(sym->sexp constructor) ,@(map unparse-type operands))))
      ((tvariable->type^ tvar)
       (sym->sexp (tvariable->sym tvar)))
      ((unknown->type^)
       '(*unknown*)))))

;; Parse a type schema (generic (I*) T)
;;
(define parse-schema
  (lambda (sexpr)
    (match sexpr
      (`(generic (,@names) ,type)
       (let ((names (map (lambda (name)
                           (match name
                             ((sym->sexp^ name) name)
                             (_ (error "invalid type schema parameter" name))))
                         names)))
         (let ((tvars (map new-tvariable names)))
           (make-schema tvars
                        (substitute-for-names (map tvariable->type tvars)
                                              names
                                              (parse-type type))))))
      (_ (make-schema (null) (parse-type sexpr))))))
```

```
(define substitute-for-names
  ; substitute-for-names is a kludge, to be used only by initialization
  ; code.  Other ways to do this: (1) change the type parser to take an
  ; environment argument; (2) generalize substitute-into-type so that it
  ; can substitute for either names or tvars; (3) change the
  ; representation of schemas so that the generic variables in the type
  ; are not tvars but rather names.
  (lambda (types names type)
    (match type
      ((tvariable->type^ _) type)  ;shouldn't happen
      ((base->type^ name)
       (letrec ((loop (lambda (ts ns)
                        (if (null? ts)
                            type
                            (if (same-name? name (car ns))
                                (car ts)
                                (loop (cdr ts) (cdr ns))))))))
         (loop types names)))
      ((compound->type^ c args)
       (compound->type c (map (lambda (arg)
                                (substitute-for-names types names arg))
                              args)))
      (_ (error "this shouldn't happen" type)))))

(define unparse-schema
  (lambda (s)
    (match s
      ((make-schema^ tvars type)
       `(generic (,@(map sym->sexp (map tvariable->sym tvars)))
                 ,(unparse-type type))))))
```

C.14 compiler/system.fx

The contents of the file compiler/system.fx:

```
;; -*- Mode: Scheme; -*-

;;
;; System routines, including garbage collector, used by DLX simulator
;;

;; System routines (which don't live in simulated memory) have
;; negative addresses so we can take their addresses, etc, and not get
;; confused.
```

```
(define enter-system-routine-labels
  (lambda ()
    (enter-label "_ALLOC" -4)
    (enter-label "_SAVE" -8)
    (enter-label "_RESTORE" -12)
    (enter-label "__EXIT" -16)
    (enter-label "_PUTCHAR" -20)
    (enter-label "_SYM2STRING" -24)
    (enter-label "_SALLOC" -28)
    (enter-label "_SFREE" -32)
    (enter-label "_BZERO" -36)))

(define system-routine
  ;; Call sys routines by passing calling this function w/ routine's addr
  (lambda (funcnumber)
    (cond ((= funcnumber -4) (allocate-block-of-memory))
          ((= funcnumber -8) (save-regs-into-frame))
          ((= funcnumber -12) (restore-regs-from-frame))
          ((= funcnumber -16) (done-emulating))
          ((= funcnumber -20) (putchar))
          ((= funcnumber -24) (sym2string))
          ((= funcnumber -28) (stack-allocate-block))
          ((= funcnumber -32) (stack-free-block))
          ((= funcnumber -36) (zero-block))
          (else (error "unknown system routine called.")))))

;; Here are the system routines.  They're basically the same as the
;; versions in ~microFX/runtime (though some of the names may have
;; changed through negligence).

;; _SYM2STRING
(define sym2string
  ;; Turn a symbol into a vector of characters
  (lambda ()
    (letrec ((str (get-mem (get-reg ATEMP)))
             (fill-block
              (lambda (i)
                (if (>= i (string-length str))
                    the-unit
                    (begin (set-slot! (get-reg ATEMP) i
                                      (itag (char->int (string-ref str i))))
                           (fill-block (+ i 1)))))))
      (begin (set-reg! ATEMP (itag (string-length str)))
             (allocate-block-of-memory)
             (fill-block 0)))))

;; _PUTCHAR
(define putchar (lambda () (display (int->char (de-itag (get-reg ATEMP))))))

;; __EXIT
(define done-emulating (lambda () (:= *halt-emulate?* #t)))
```

```
;; _SAVE
(define save-regs-into-frame
  (lambda ()
    (letrec ((f (get-reg FP))
             (loop (lambda (reg slot)
                     (if (= reg 30)
                         the-unit
                         (begin (set-slot! f slot (get-reg reg))
                                (loop (+ 1 reg) (+ 1 slot)))))))
      (loop 6 4))))

;; _RESTORE
(define restore-regs-from-frame
  (lambda ()
    (letrec ((f (get-reg FP))
             (loop (lambda (reg slot)
                     (if (= reg 30)
                         the-unit
                         (begin (set-reg! reg (get-slot f slot))
                                (loop (+ 1 reg) (+ 1 slot)))))))
      (loop 6 4))))

;; For the allocator and garbage collector, see the commentary in
;; ~microFX/runtime/alloc.s.  This version is sufficiently similar
;; that those comments should apply here.

;; _ZBLOCK
(define zero-block
  (lambda ()
    (letrec ((loop (lambda (blk len) (if (= len 0)
                                          the-unit
                                          (begin (set-slot! blk (- len 1) 0)
                                                 (loop blk (- len 1)))))))
      (loop (get-reg ATEMP) (de-itag (get-slot (get-reg ATEMP) -1))))))

;; _SALLOC
(define stack-allocate-block
  (lambda ()
    (let* ((nslots (de-itag (get-reg ATEMP)))
           (new-sp (- (get-reg SP) (+ (* nslots 4) 4))) ; Extra slot for size
           (stack-size (- (- (^ *entire-memory-size*) (quotient new-sp 4)) 1))
           (new-blk (+ new-sp 5))) ; Tagged ptr to new GC block
      (begin
        (if (> stack-size (^ *max-stack-size*))
            (:= *max-stack-size* stack-size)
            the-unit)
        (if (> stack-size (^ *stack-size*))
          (error "Stack overflow.")
          (begin
            (:= *total-allocation* (+ 1 (+ (^ *total-allocation*) nslots)))
            (set-slot! new-blk -1 (get-reg ATEMP)) ; Size
            (set-reg! ATEMP new-blk)
            (zero-block)
            (set-reg! SP new-sp)))))))
```

```
;; _SFREE
(define stack-free-block
  (lambda ()
    (let* ((nslots (de-itag (get-slot (get-reg ATEMP) -1)))
           (new-sp (+ (get-reg SP) (+ (* nslots 4) 4))))
      (if (>= new-sp (* (^ *entire-memory-size*) 4))
          (error "Stack underflow.")
          (set-reg! SP new-sp)))))

;; _ALLOC
(define allocate-block-of-memory
  (lambda ()
    (let* ((nslots (de-itag (get-reg ATEMP)))
           (p (allocate-raw-block-of-memory (+ nslots 1))))
      (set-slot! p -1 (get-reg ATEMP))          ; (tagged)
      (set-reg! ATEMP p)
      (zero-block))))

(define allocate-raw-block-of-memory
  (lambda (nslots)
    (let ((block (get-reg HP)))
      (begin
        (set-reg! HP (+ block (* nslots 4)))    ; 4 bytes per slot
        (if (<= (get-reg HP) (^ *this-semispace-end*))
            (begin
              (:= *total-allocs* (+ (^ *total-allocs*) 1))
              (:= *total-allocation* (+ (^ *total-allocation*) nslots))
              block)
            (begin
              (let ((stack-atemp (get-reg ATEMP)))
                (interpreter-gc)
                (set-reg! ATEMP stack-atemp))
              (:= *num-gcs* (+ (^ *num-gcs*) 1))
              (:= *gc-words-copied*
                  (+ (^ *gc-words-copied*)
                     (quotient (- (get-reg HP) (^ *this-semispace*)) 4)))
              (if (> (+ (get-reg HP) (* nslots 4)) (^ *this-semispace-end*))
                  (error "out of memory!")
                  (allocate-raw-block-of-memory nslots))))))))

(define stack-depth
  (letrec ((loop (lambda (p count)
                   (if (even? p)
                       count
                       (loop (get-slot p 0) (+ count 1))))))
    (lambda () (loop (get-reg FP) 0))))

(define interpreter-gc
  (lambda ()
    (begin
      (if (^ *noisy-gc*)
          (begin (newline)
                 (display "- - - - - - - Beginning garbage-collection"))
          the-unit)
```

```
        ;; Create frame on stack to save old values of registers
        (set-reg! ATEMP (itag FrameSize))
        (stack-allocate-block)
        (set-slot! (get-reg ATEMP) 0 (get-reg FP))
        (set-reg! FP (get-reg ATEMP))
        (set-slot! (get-reg FP) 3 (get-reg ENV))
        (save-regs-into-frame)

        ;; flip the semispaces
        (let ((old-start (^ *this-semispace*))
              (old-end (^ *this-semispace-end*)))
          (begin (:= *this-semispace* (^ *other-semispace*))
                 (:= *other-semispace* old-start)
                 (:= *this-semispace-end* (^ *other-semispace-end*))
                 (:= *other-semispace-end* old-end)))
        (set-reg! HP (^ *this-semispace*))

        ;; scan the root set
        (set-reg! FP (maybe-copy (get-reg FP)))
        (set-reg! VAL (maybe-copy (get-reg VAL)))

        ;; Restore registers from stack frame and pop stack frame
        (restore-regs-from-frame)
        (set-reg! ATEMP (get-reg FP))
        (set-reg! ENV (get-slot (get-reg FP) 3))
        (set-reg! FP (get-slot (get-reg FP) 0))
        (stack-free-block)

        (if (^ *noisy-gc*)
            (begin
              (newline)
              (display "- - - - - - - Garbage-collection done: ")
              (display (quotient (- (^ *this-semispace-end*) (get-reg HP)) 4))
              (display "/")
              (display (^ *semispace-size*))
              (display " words free."))
            the-unit))))

(define in-thisspace?
  (lambda (p) (and (>= p (^ *this-semispace*))
                   (< p (^ *this-semispace-end*)))))

(define in-otherspace?
  (lambda (p) (and (>= p (^ *other-semispace*))
                   (< p (^ *other-semispace-end*)))))

(define in-stack?
  (lambda (p) (> p (get-reg SP))))
```

```
(define maybe-copy
  ;; Copy heap blocks into new heap space, returning new result
  (lambda (p)
    (cond ((even? p) p) ; don't copy atoms
          ((in-thisspace? p) p) ; optimization: obj's in thisS already scanned
          ((not (in-otherspace? p))
           (begin (scan-transitively p) p)) ; only scan blocks not in old space
          ((odd? (get-slot p -1)) (get-slot p -1))        ; return forward addr
          (else ;; ok, we've got a live one.
           (let ((newp (get-reg HP))
                 (nslots (+ (de-itag (get-slot p -1)) 1))) ; untagged wordcount
             (begin (set-reg! HP (+ newp (* nslots 4)))
                    (set-slot! newp -1 (get-slot p -1)) ; size field
                    (copy-block newp p (- nslots 1))    ; data fields
                    (set-slot! p -1 newp)               ; set forward ptr.
                    (scan-transitively newp)
                    newp))))))

(define copy-block      ; copies from slot 0 to slot "slot - 1", inclusive.
  (lambda (new old slot)
    (if (<= slot 0)
        the-unit
        (begin (set-slot! new (- slot 1) (get-slot old (- slot 1)))
               (copy-block new old (- slot 1))))))

(define scan-transitively
  (lambda (p)
    (letrec ((loop (lambda (slot)
                     (if (< slot 0)
                         the-unit
                         (begin
                           (set-slot! p slot (maybe-copy (get-slot p slot)))
                           (loop (- slot 1)))))))
      (loop (- (de-itag (get-slot p -1)) 1)))))
```

C.15 compiler/table.fx

The contents of the file compiler/table.fx:

```
;;
;;  polymorphic symbol-tables (compile-time environments)
;;

;; There are three components to this package: a function to create an
;; empty environment "(mk-empty-env empty-function)", a function to create a
;; "binder" (i.e. a function to enter key/value pairs into the table)
;; "(mk-binder equality-comparator)", and a polymorphic lookup function
;; "(lookup key env)".
```

```
;;;  types used in the comments below:
;;;;
;;;;; empty-flag : empty-type
;;;;; empty-type == value-type
;;;;; key : key-type
;;;;; env : key-type -> value-type




;;;;; mk-empty-env : (poly (key-type value-type)
;;;;;                   (-> ((-> (key-type) value-type)) env))
;;;;;
(define mk-empty-env
  (lambda (empty-fn) (lambda (key) (empty-fn key))))




;;;;; mk-binder : (poly (key-type value-type)
;;;;;               (-> ((-> (key-type key-type) bool))
;;;;;                 (-> (key-type value-type env) env)
;;;;;  here, "env" is a macro, borrowing the definitions of key-type and
;;;;;  value-type from the poly params.
;;;;;
(define mk-binder
  (lambda (key=?)
    (lambda (key value env)
      (lambda (new-key)
        (if (key=? key new-key)
            value
            (lookup new-key env))))))



;;;;; lookup : (poly (key-type value-type)
;;;;;             (-> (key-type (-> (key-type) value-type))
;;;;;              value-type))
;;;;;
(define lookup (lambda (key env) (env key)))
```

```
;;
;; a handy utility...
;;
(define list2env
  (lambda (empty-fn key=? pairs)
    (letrec ((bind (mk-binder key=?))
             (l2e (lambda (l t)
                    (if (null? l)
                        t
                        (l2e (cdr l)
                             (bind (car (car l)) (cadr (car l)) t))))))
      (l2e pairs (mk-empty-env empty-fn)))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Sample to run through microfx to check typing... (yup, it worked)
;
;(run
;  '(letrec ((mk-empty-env (lambda (empty-flag) (lambda (key) empty-flag)))
;            (mk-binder (lambda (key=?)
;                         (lambda (key value env)
;                           (lambda (new-key)
;                             (if (key=? key new-key)
;                                 value
;                                 (lookup new-key env))))))
;            (lookup (lambda (key env) (env key))))
;           (let (
;                 (empty1 (mk-empty-env -1)) (bind1 (mk-binder =))
;                 (empty2 (mk-empty-env -2)) (bind2 (mk-binder sym=?))
;                 )
;                (begin
;                 (+
;                  (lookup 1 (bind1 2 100 (bind1 1 99 empty1)))
;                  (lookup (symbol foo)
;                          (bind2 (symbol foo) 500
;                                 (bind2 (symbol bar) 600 empty2))))
;                 )
;                )))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

C.16 compiler/toplevel.fx

The contents of the file compiler/toplevel.fx:

```
;; -*- Mode: Scheme; Package: SCHEME -*-

(define *verbose-flag* (ref #f))        ;; makes the interpreter noisy...
(define *silent-flag* (ref #f))         ;; makes the interpreter SILENT.
```

```
;; parse a microFX s-expression
;;
(define test-parse
  (lambda (sexp)
    (pprint-exp (parse sexp))))

;; parse a microFX s-expression
;; (should reproduce input)
;;
(define test-parse-simple
  (lambda (sexp)
    (unparse (parse sexp))))


;; type-check a microFX s-expression
;;
(define check
  (lambda (e)
    (unparse-type (reconstruct-top (parse e)))))

;; type-check a microFX s-expression and display the expression tree
;; annotated with the reconstructed type information.
;;
(define show-type-check
  (lambda (e)
    (let ((parse-tree (parse e)))
      (begin (reconstruct-top parse-tree)
             (pprint-exp-types parse-tree)))))


;; compile expression to icode.
;;
(define itest-compile
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (newline) (display "Type: ") (write (unparse-type type)) (newline)
        (generate-icode exp)
        (display-icode-list)))))



;; compile expression to optimized icode.
;;
(define otest-compile
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (newline) (display "Type: ") (write (unparse-type type)) (newline)
        (generate-icode exp)
        (optimize-icode)
        (display-icode-list)))))
```

```
;; compile expression to ocode (prints assemblycode).
;;
(define test-compile
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (newline) (display "Type: ") (write (unparse-type type)) (newline)
        (generate-icode exp)
        (optimize-icode)
        (generate-ocode)
        (newline) (display "Object code: ") (newline)
        (display-ocode-list (current-output-port))
        (newline)))))



;; compile to ocode, then interpret ocode.
;;
(define run
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (if (^ *silent-flag*)
            the-unit
            (begin (newline)
                   (display "Type: ") (write (unparse-type type)) (newline)))
        (generate-icode exp)
        (optimize-icode)
        (generate-ocode)
        (if (^ *silent-flag*)
            the-unit
            (begin (newline) (display "Running: ") (newline)))
        (init-emulator)
        (rerun)
        (if (^ *silent-flag*)
            the-unit
            (begin (newline) (show-stats)))
        (extract-value (get-reg VAL) type)))))
```

```
;; compile to ocode, then interpret ocode. free-up spare memory first!
;; (worse for debugging, but better for automatic testing.)
;;
(define run-m
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (if (^ *silent-flag*)
            the-unit
            (begin (newline)
                   (display "Type: ") (write (unparse-type type)) (newline)))
        (generate-icode exp)
        (optimize-icode)
        (generate-ocode)
        (if (^ *silent-flag*)
            the-unit
            (begin (newline) (display "Running: ") (newline)))
        (:= icode-list (null))
        (init-emulator)
        (:= icode-to-be-emitted (null))
        (:= library-icode (null))
        (rerun)
        (if (^ *silent-flag*)
            the-unit
            (begin (newline) (show-stats)))
        (extract-value (get-reg VAL) type)))))


;; like run, but verbose (prints instruction stream as it executes).
;;
(define runv
  (lambda (sexpr)
    (let ((old-vflag (^ *verbose-flag*)))
      (begin
        (:= *verbose-flag* #t)
        (let ((retval (run sexpr)))
          (begin (:= *verbose-flag* old-vflag)
                 retval))))))


;; print out various statistics about the run (currently only gc stats).
;;
(define show-stats
  (lambda ()
    (begin
      (newline)
      (display "#gc's=") (display (^ *num-gcs*))
      (display " words copied by gc=") (display (^ *gc-words-copied*))
      (display " words allocated=") (display (^ *total-allocation*))
      (display " total-allocs=") (display (^ *total-allocs*))
      (newline))))
```

```
;; compile for dlxsim into the file "fx.s".
;;
(define fx
  (lambda (sexpr) (fx-with-outname sexpr "fx.s")))



;; compile for dlxsim into a named file.
;;
(define fx-with-outname
  (lambda (sexpr outname)
    (let ((prologname (string-append compiler-directory "/prolog.code"))
          (epilogname (string-append compiler-directory "/epilog.code"))
          (old-pp (^ *rr-pretty*)))
      (let* ((exp (parse sexpr))
             (type (reconstruct-top exp))
             (fout (open-output-file outname)))
        (begin
          (:= *rr-pretty* #f) ; No pprinting for DLX-ASM output
          (newline) (display "Type: ") (write (unparse-type type)) (newline)

          ;; Compile program
          (generate-icode exp)
          (optimize-icode)
          (generate-ocode)

          ;; Copy prolog to output file
          (newline fout)
          (let ((prolog (open-input-file prologname)))
            (copy-input-stream-to-output-stream prolog fout)
            (close-input-port prolog))

          ;; write compiled code (as text) to output file
          (display-ocode-list fout)

          ;; Output format string to print the result (see printf.s in runtime)
          (display "RESULT_FORMAT:" fout) (newline fout)
          (display "    .ascii    " fout)
          (write (type-to-printf-format type) fout)
          (newline fout)
          (display "    .byte     0x0a,0" fout) (newline fout)
          (display "    .align     2" fout)     (newline fout)

          ;; Copy epilog to output file
          (newline fout)
          (let ((epilog (open-input-file epilogname)))
            (copy-input-stream-to-output-stream epilog fout)
            (close-input-port epilog))
          (close-output-port fout)
```

```
          (display "Object code has been written to file ") (display outname)
          (newline)
          (:= *rr-pretty* old-pp))))))


(define listing
  (lambda () (do-listing (^ ocode-list) "fx.asm")))

(define do-listing
  (lambda (l outname)
    (letrec ((fout (open-output-file outname))
             (loop (lambda (pc l)
                     (if (null? l)
                         the-unit
                         (let ((new-pc (match (car l)
                                         ((ocode^ 'labeldef _) pc)
                                         ((ocode^ 'stringdef _) (+ pc 4))
                                         (_ (+ pc 4)))))
                           (begin
                             (newline fout)
                             (if (= pc new-pc)
                                 the-unit
                                 (begin
                                   (if (< pc 1000) (display "0" fout) the-unit)
                                   (if (< pc 100) (display "0" fout) the-unit)
                                   (if (< pc 10) (display "0" fout) the-unit)
                                   (display pc fout) (display ": " fout)))
                             (display (unparse-ocode (car l)) fout)
                             (loop new-pc (cdr l)))))))))
      (begin (loop 0 l)
             (close-output-port fout)
             (newline)
             (display "Listing has been written to file ") (display outname)
             (newline)))))


;; some shorthand...
;;
(define tc test-compile)
(define ic itest-compile)
(define oc otest-compile)



;; print-out the list of icode.
;;
(define display-icode-list
  (lambda ()
    (begin
      (newline) (display "Icode: ") (newline)
      (for-each display-icode (^ icode-list))
      (newline))))
```

```
;; print-out the list of ocode.
;;
(define display-ocode-list
  (lambda (strm)
    (newline) (display "Code: ") (newline)
    (for-each
        (lambda (x) (print-one-instruction x strm))
        (^ ocode-list))))
```

C.17 compiler/ty_recon.fx

The contents of the file compiler/ty_recon.fx:

```
; -*- Mode: Scheme; -*-
;;;
;;; modified 11/14/90 by jwo for mini-fx(90) version
; Contains the occurs-check fix

; Correction for future: when printing out type
; clashes, should substitute types.


; Most, but not all, of a type reconstruction program.

; Type reconstruction

(define reconstruct-top
  (lambda (e)
    (begin
      (reset-tvariable-counter!)
      (reconstruct e standard-type-environment))))
```

```
(define reconstruct
  (lambda (exp tenv)
    (match exp
      ((variable->exp^ type-ptr var)
       (memoize-type type-ptr (reconstruct-variable var tenv)))
      ((bool->exp^ type-ptr _)
       (memoize-type type-ptr boolean-type))
      ((int->exp^ type-ptr _)
       (memoize-type type-ptr integer-type))
      ((char->exp^ type-ptr _)
       (memoize-type type-ptr character-type))
      ((string->exp^ type-ptr _)
       (memoize-type type-ptr string-type))
      ((sym->exp^ type-ptr _)
       (memoize-type type-ptr symbol-type))
      ((conditional->exp^ type-ptr test con alt)
       (memoize-type type-ptr (reconstruct-conditional test con alt tenv)))
      ((begin->exp^ type-ptr exprs)
       (memoize-type type-ptr (reconstruct-begin exprs tenv)))
      ((abstraction->exp^ type-ptr formals body)
       (memoize-type type-ptr (reconstruct-abstraction formals body tenv)))
      ((combination->exp^ type-ptr op args)
       (memoize-type type-ptr (reconstruct-combination op args tenv)))
      ((binder->exp^ type-ptr defs body)
       (memoize-type type-ptr (reconstruct-binder defs body tenv)))
      ((recursion->exp^ type-ptr defs body)
       (memoize-type type-ptr (reconstruct-recursion defs body tenv)))))))

(define set-type!
  (lambda (exp type)
    (match exp
      ((variable->exp^ type-ptr _)         (memoize-type type-ptr type))
      ((bool->exp^ type-ptr _)             (memoize-type type-ptr type))
      ((int->exp^ type-ptr _)              (memoize-type type-ptr type))
      ((char->exp^ type-ptr _)             (memoize-type type-ptr type))
      ((string->exp^ type-ptr _)           (memoize-type type-ptr type))
      ((sym->exp^ type-ptr _)              (memoize-type type-ptr type))
      ((conditional->exp^ type-ptr _ _ _) (memoize-type type-ptr type))
      ((begin->exp^ type-ptr _)            (memoize-type type-ptr type))
      ((abstraction->exp^ type-ptr _ _)    (memoize-type type-ptr type))
      ((combination->exp^ type-ptr _ _)    (memoize-type type-ptr type))
      ((binder->exp^ type-ptr _ _)         (memoize-type type-ptr type))
      ((recursion->exp^ type-ptr _ _)      (memoize-type type-ptr type)))))

(define memoize-type
  (lambda (type-ptr type)
    (begin (:= type-ptr type)
           type)))
```

```
(define reconstruct-variable
  (lambda (var tenv)
    (let ((tvar-or-schema (tlookup tenv var)))
      (match tvar-or-schema
        ((tvar->tvar-or-schema~ tvar)
         (tvariable->type tvar))
        ((schema->tvar-or-schema~ schema)
         (instantiate-schema schema))))))

(define reconstruct-conditional          ; if
  (lambda (test con alt tenv)
    (begin (unify! (reconstruct test tenv) boolean-type)
           (let ((con-type (reconstruct con tenv))
                 (alt-type (reconstruct alt tenv)))
             (begin (unify! con-type alt-type)
                    con-type)))))

(define reconstruct-begin
  (lambda (exprs tenv)
    (begin (map (lambda (exp) (reconstruct exp tenv)) exprs)
           (expression-type (car (list-tail exprs (- (length exprs) 1)))))))

(define reconstruct-abstraction          ; lambda
  (lambda (vars body tenv)
    (let ((new-tvars (map new-tvariable vars)))
      (make-arrow-type
       (map tvariable->type new-tvars)
       (reconstruct body
                    (extend-by-tvariables tenv vars new-tvars))))))

(define reconstruct-combination          ; call
  (lambda (op args tenv)
    (let ((arg-types (map (lambda (arg) (reconstruct arg tenv)) args))
          (result-type (tvariable->type (new-tvariable (symbol result)))))
      (begin (unify! (reconstruct op tenv)
                     (make-arrow-type arg-types result-type))
             result-type))))


(define reconstruct-binder               ; let
  (lambda (defs body tenv)
      (reconstruct body
                   (extend-by-schemas
                    tenv
                    (map definition-name defs)
                    (map (lambda (binding) (compute-schema (reconstruct binding tenv)
                                                           tenv))
                         (map definition-value defs))))))
```

```
(define reconstruct-recursion          ; letrec
  (lambda (defs body tenv)
    (let* ((names (map definition-name defs))
           (tvars (map new-tvariable names))
           (dummy-tenv (extend-by-tvariables tenv names tvars))
           (types (map (lambda (def)
                         (reconstruct (definition-value def) dummy-tenv))
                       defs))
           (new-tenv (extend-by-schemas
                      tenv names
                      (map (lambda (t) (compute-schema t tenv)) types))))
      (begin (for-each-2 unify! (map tvariable->type tvars) types)
             (reconstruct body new-tenv)))))

; Note: the use of UNIFY!-LIST rather than FOR-EACH-2 fails to
; correctly type (or find a type error in) the following example:
; (check '(letrec ((a (lambda () 3))
;                  (b (if (a) 1 2)))
;           4))


; Type schemas
(define compute-schema                  ;Function GEN from handout
  (lambda (type tenv)
    (make-schema (generic-tvariables type tenv)
                 type)))

; NOTE: generic-tvariables looks not only at tvariables in the
; given type, but also at tvariables in the leaves of
; the fully unwound version of the given type.  This interacts with
; a similar unwinding at instantiation time to appropriately handle
; generalization.  There is potential confusion in that the returned
; list may contain types that are not manifestly in TYPE but are in
; the fully unwound tree associated with it.

(define generic-tvariables              ;Compute FTV(type) - FTE(tenv)
  (lambda (type tenv)
    (match (prune type)
      ((tvariable->type~ tvar)
       (if (generic-tvariable? tvar tenv)
           (list tvar)
           (null)))
      ((compound->type~ _ operands)
       (letrec ((loop (lambda (ops tvars)
                        (if (null? ops)
                            tvars
                            (loop (cdr ops)
                                  (union (generic-tvariables (car ops) tenv)
                                         tvars))))))
         (loop operands (null))))
      ((base->type~ _) (null))
      (_ (error "this shouldn't happen" type)))))
```

```
(define (union l1 l2)
  (cond ((null? l1) l2)
        ((null? l2) l1)
        ((in-tvariable-list? (car l1) l2) (union (cdr l1) l2))
        (else (cons (car l1) (union (cdr l1) l2)))))

; [The following use of MEMQ is a Mini-FX type error and an
; abstraction violation, but it works & is fast.]

(define in-tvariable-list? memq)

; Instantiate a type schema on a fresh set of type variables.
; [This corresponds to Cardelli's "FreshType".]

(define instantiate-schema
  (lambda (schema)
    (substitute-into-type
     (map (lambda (g) (tvariable->type (new-tvariable (tvariable-id g))))
          (schema-generics schema))
     (schema-generics schema)
     (schema-type schema))))


; [The following corresponds to Cardelli's "Fresh"; note the call to prune.]

; Note that this unwinds TYPE out to the leaves when doing the substitution;
; this guarantees that we don't miss any substitutions because type itself
; isn't fully unwound.

(define substitute-into-type
  (lambda (types tvars type)
    (let ((type (prune type)))
      (match type
        ((tvariable->type~ tvar)
         (letrec ((loop (lambda (ts tvars)
                          (if (null? ts)
                              type
                              (if (same-tvariable? tvar (car tvars))
                                  (car ts)
                                  (loop (cdr ts) (cdr tvars)))))))
           (loop types tvars)))
        ((base->type~ _) type)
        ((compound->type~ c args)
         (compound->type c (map (lambda (arg)
                                  (substitute-into-type types tvars arg))
                                args)))
        (_ (error "this shouldn't happen" type))))))
```

```
; Type environments.
;
;  Environments can be extended in either of two ways:
;    extend-by-tvariables  should be used by lambda and letrec to bind
;      variables to type variables
;    extend-by-schemas  should be used by let and letrec to bind variables
;      to type schemas
;
;  Once constructed, there are two operations one can perform on a
;  type environment:
;    tlookup : tenv * var -> (tvar + schema)
;      does the usual thing.
;    generic-tvariable? : tvar * tenv -> bool
;      returns true iff tvar is not free in the type of any var bound in tenv.

(define-datatype type-environment
  (make-type-env tlookup-proc generic-tvariable?-proc))

(define (tenv-lookup te)
  (match te
    ((make-type-env~ lookup generic?) lookup)))

(define (tenv-generic? te)
  (match te
    ((make-type-env~ lookup generic?) generic?)))

(define extend-by-tvariables
  (lambda (outer-tenv vars tvars)
    (extend-tenv
     outer-tenv
     vars
     (map tvar->tvar-or-schema tvars)
     (lambda (tvar)
       ;; tvar is an unconstrained type variable.
       (letrec ((loop (lambda (tvars)
                        (if (null? tvars)
                            (generic-tvariable? tvar outer-tenv)
                            (if (occurs-in-type? tvar
                                                 (tvariable->type (car tvars)))
                                ;; (same-tvariable? tvar (car tvars))
                                #f
                                (loop (cdr tvars)))))))
         (loop tvars))))))

(define extend-by-schemas
  (lambda (outer-tenv vars schemas)
    (extend-tenv outer-tenv
                 vars
                 (map schema->tvar-or-schema schemas)
                 (lambda (tvar)
                   (generic-tvariable? tvar outer-tenv)))))
```

```
(define extend-tenv                       ;Students' code should not call this
  (lambda (outer-tenv vars typas generic-tvariable?-proc)
    (make-type-env
     (lambda (var)
       (letrec ((loop (lambda (vars typas)
                        (if (null? vars)
                            (tlookup outer-tenv var)
                            (if (same-variable? var (car vars))
                                (car typas)
                                (loop (cdr vars) (cdr typas)))))))
         (loop vars typas)))
     generic-tvariable?-proc)))

(define empty-type-environment
  (make-type-env
   (lambda (var) (error "unbound variable" var))
   (lambda (tvar) #t)))

(define tlookup
  (lambda (tenv var)
    ((tenv-lookup tenv) var)))

(define same-variable? sym=?)

(define generic-tvariable?
  (lambda (tvar tenv)
    ((tenv-generic? tenv) tvar)))

; Proving the correctness of this implementation of GENERIC-TVARIABLE?
; is tricky.


; A type variable is implemented as a record that contains a ref.  The
; global substitution is realized as the collective contents of the
; refs for all type variables.

(define-datatype tvariable
  (make-tvariable sym int (refof type)))        ; id gennum ref

(define tvariable-id
  (lambda (tvar)
    (match tvar
      ((make-tvariable^ id _ _) id))))

(define tvariable-ref
  (lambda (tvar)
    (match tvar
      ((make-tvariable^ _ _ r) r))))

(define tvariable-counter (ref 0))

(define reset-tvariable-counter!
  (lambda () (:= tvariable-counter 0)))
```

```
(define new-tvariable
  (lambda (id)
    (begin (:= tvariable-counter (+ (^ tvariable-counter) 1))
           (make-tvariable id (^ tvariable-counter) (ref unknown-type)))))

(define (tvariable-binding tvar)
  (^ (tvariable-ref tvar)))

(define extend-substitution!
  (lambda (tvar binding)
    (begin (:= (tvariable-ref tvar) binding)
           #t)))

(define same-tvariable?
  (lambda (tvar1 tvar2)
    (same-ref? (tvariable-ref tvar1) (tvariable-ref tvar2))))

(define unknown-type (unknown->type))

(define tvariable->sym
  (lambda (tvar)
    (match tvar
      ((make-tvariable^ id gennum _)
       (string->sym (string-append (string-append "?" (sym->string id))
                                   (string-append "-" (int->string gennum))))))))


; Unification
;  Has side effects.
;  Generates an error if there is no unification.

(define unify!
  (lambda (type1 type2)
    (if (unify!-internal type1 type2)
        the-unit
        (error "type clash" (unparse-type type1) (unparse-type type2)))))
```

```
(define unify!-internal
  (lambda (type1 type2)
    (let ((type1 (prune type1))
          (type2 (prune type2)))
      ;; Now if a type is a variable, it will be unbound
      (match type1
        ((tvariable->type^ v1)
         (match type2
           ((tvariable->type^ v2)
            (if (same-tvariable? v1 v2)
                #t
                (extend-substitution! v1 type2)))
           (_
            (if (occurs-in-type? v1 type2)
                #f                     ;Circularity
                (extend-substitution! v1 type2)))))
        ((base->type^ c1)
         (match type2
           ((tvariable->type^ v2)
            (extend-substitution! v2 type1))
           ((base->type^ c2)
            (same-name? c1 c2))
           (_ #f)))
        ((compound->type^ con1 args1)
         (match type2
           ((tvariable->type^ v2)
            (if (occurs-in-type? v2 type1)
                #f
                (extend-substitution! v2 type1)))
           ((compound->type^ con2 args2)
            (if (same-constructor? con1 con2)
                (unify!-list args1 args2)
                #f))
           (_ #f)))))))

(define unify!-list
  (lambda (types1 types2)
    (if (null? types1)
        (null? types2)
        (if (null? types2)
            #f
            (if (unify!-internal (car types1) (car types2))
                (unify!-list (cdr types1) (cdr types2))
                #f)))))

; Chase substitutions of tvariables until either a non-tvariable or an
; unbound tvariable is found.
```

```
(define prune
  (lambda (type)
    (match type
      ((tvariable->type^ tvar)
       (match (tvariable-binding tvar)
         ((unknown->type^) type)
         (other-type (prune other-type))))
      (_ type))))

; Prevent circular substitutions.

(define occurs-in-type?
  (lambda (tvar type)
    (match (prune type)
      ((tvariable->type^ tvar2)
       ;; prune has guaranteed that tvar2 is unbound
       (same-tvariable? tvar tvar2))
      ((compound->type^ c args)
       (letrec ((loop (lambda (args)
                        (if (null? args)
                            #f
                            (or (occurs-in-type? tvar (car args))
                                (loop (cdr args)))))))
         (loop args)))
      (_ #f))))

(define the-unit (:= (ref 0) 0))
```

D μFX/DLX Run-time Implementation

This appendix contains a snapshot as of February 12, 1992 of the 8 source files which implement the runtime system. All of these files are available via FTP.

The files included in this appendix are as follows:

Filename	Module	Purpose
runtime/Makefile	Support	Makefile for processing runtime DLX code
runtime/alloc.s	Runtime	DLX Memory allocation and garbage collection
runtime/epilog.s	Output	DLX code to follow compiled code
runtime/frames.s	Runtime	Runtime code for register save and restore
runtime/lib.s	Runtime	Miscellaneous runtime primitives
runtime/macros.h	Support	Macros used in prolog.s and epilog.s
runtime/printf.s	Runtime	DLX code implementing printf utility
runtime/prolog.s	Output	DLX code to precent compiled code

The index at the end of this document contains entries for procedures, shared variables, and runtime entry points.

D.1 runtime/Makefile

The contents of the file runtime/Makefile:

```
CPP=cc -E -traditional
#CPP=gcc -E -traditional
#CPP=/lib/cpp -P

all: ../epilog.code ../prolog.code

../epilog.code: macros.h epilog.s alloc.s frames.s printf.s lib.s
	$(CPP) epilog.s \
		| sed -g 's/;.*$$//' \
		| sed -g 's/        /  /g' \
		| egrep -v '^[  ]*$$|^#.*|
' \
		> ../epilog.code

../prolog.code: macros.h prolog.s
	$(CPP) prolog.s \
		| sed -g 's/;.*$$//' \
		| sed -g 's/        /  /g' \
		| egrep -v '^[  ]*$$|^#.*|
' \
		> ../prolog.code
```

D.2 runtime/alloc.s

The contents of the file runtime/alloc.s:

```
;; alloc.s

;;  Here lies the memory allocator and garbage collector.

;;  If you have to change it, then be careful, and TEST OFTEN (that
;;  means REALLY often), because it's a real pain to debug if it gets
;;  broken.


;;  LEAVE ALL THIS DATA  TOGETHER (for addressability)!
Memory_description:
	.word   0	; Dummy value
;; Bounds on memory use
_semispace_size:
	.word   2048
_stack_size:
	.word   2048
```

```
;; Pointers that describe the two semispaces.
_this_semispace:
        .word   0
_this_semispace_end:
        .word   0
_other_semispace:
        .word   0
_other_semispace_end:
        .word   0

;; Statistics
_num_gcs: ; count of gc's performed
        .word   0
_gc_words_copied: ; number of words scanned by all gc's
        .word   0
_total_allocation: ; total #words allocated (ever)
        .word   0
_total_allocs:
        .word   0
_max_stack_size:
        .word   0

;; Flags and misc...
_just_did_a_gc:
        .word   0

;; Address above locations by offsets from Memory_description
#define semispace_size          _semispace_size-Memory_description
#define stack_size              _stack_size-Memory_description
#define this_semispace          _this_semispace-Memory_description
#define this_semispace_end      _this_semispace_end-Memory_description
#define other_semispace         _other_semispace-Memory_description
#define other_semispace_end     _other_semispace_end-Memory_description
#define num_gcs                 _num_gcs-Memory_description
#define gc_words_copied         _gc_words_copied-Memory_description
#define total_allocation        _total_allocation-Memory_description
#define total_allocs            _total_allocs-Memory_description
#define max_stack_size          _max_stack_size-Memory_description
#define just_did_a_gc           _just_did_a_gc-Memory_description

;; _init_runtime:
;;      Initialize runtime system.  Puts startup values in the runtime
;; system's variables and initializes SP, and HP.  Trashes ATEMP, RETADR;
;; zeros ARG0--ARG3.
_init_runtime:
        ;; Initialize memory-management registers
        lhi     SP,(TOTALMEMSIZE-4)>>16
        ori     SP,SP,(TOTALMEMSIZE-4)&0xffff
        lhi     HP,(_endprogram+1)>>16          ; tag this pointer.
        ori     HP,HP,(_endprogram+1)&0xffff
```

```
        ;; Initialize the semispace descriptions
        lhi     ARG1,(Memory_description)>>16
        ori     ARG1,ARG1,(Memory_description)&0xffff
        lw      ARG3,semispace_size(ARG1)
        slli    ARG3,ARG3,2                     ; *4 = byte count
        sw      this_semispace(ARG1),HP
        addu    ARG2,HP,ARG3
        sw      this_semispace_end(ARG1),ARG2
        sw      other_semispace(ARG1),ARG2
        addu    ARG2,ARG2,ARG3
        sw      other_semispace_end(ARG1),ARG2

        ;; Clear statistics...
        sw      num_gcs(ARG1),ZERO
        sw      gc_words_copied(ARG1),ZERO
        sw      total_allocation(ARG1),ZERO
        sw      total_allocs(ARG1),ZERO
        sw      max_stack_size(ARG1),ZERO

        or      ARG1,ARG1,ZERO
        or      ARG2,ARG1,ZERO
        or      ARG3,ARG1,ZERO
        jr      RETADR
        nop

        ;;  _ZBLOCK -- zero gc block
        ;;  On entry:
        ;;      ATEMP contains (tagged) pointer to GC block (with size
        ;;    slot filled in).
        ;;
        ;;  On return:
        ;;      Data slots of GC block are all zero.
        ;;      ATEMP still points to GC block
_ZBLOCK:
        sw      0(SP),ARG1              ; free-up a temp
        sw      -4(SP),ARG2
        lw      ARG1,-1(ATEMP)          ; load (tagged) size into ARG2
        or      ARG2,ZERO,ATEMP
zblock_loop:
        beqz    ARG1,zblock_done
        nop
        sw      3(ARG2),ZERO            ; zap a word
        addui   ARG2,ARG2,4             ; inc pointer
        j       zblock_loop
        subui   ARG1,ARG1,2             ; dec count
zblock_done:
        lw      ARG2,-4(SP)
        jr      RETADR
        lw      ARG1,0(SP)
```

```
        ;;  SALLOC -- stack allocation of GC block
        ;;  On entry:
        ;;      ATEMP contains the (tagged) number of words to allocate.
        ;;
        ;;  On return:
        ;;      ATEMP contains a (tagged) pointer to a new gc block of
        ;;    the requested size.  This block has its size slot filled
        ;;    and all data slots set to zero
        ;;      SP decremented to make room for new block
        ;;      All other regs are preserved
_SALLOC:
        subu    SP,SP,ATEMP
        subu    SP,SP,ATEMP
        subui   SP,SP,4
        sw      0(SP),ARG1              ; free-up a temp
        sw      -4(SP),ARG2             ; free-up another temp
        sw      -8(SP),ARG3             ; free-up another temp

        lhi     ARG1,(Memory_description)>>16
        ori     ARG1,ARG1,(Memory_description)&0xffff
        lw      ARG2, stack_size(ARG1)
        slli    ARG2,ARG2,2             ; *4 = byte count
        lhi     ARG3,(TOTALMEMSIZE-4)>>16
        ori     ARG3,ARG3,(TOTALMEMSIZE-4)&0xffff
        subu    ARG2,ARG3,ARG2

        slt     ARG2, SP, ARG2  ; Stack overflow?
        bnez    ARG2, stack_overflow
        nop

        addui   ARG3, SP, 5             ; Tagged ptr to new block
        sw      -1(ARG3), ATEMP         ; Save size into size slot

        ;; Increment statistics
        srli    ATEMP,ATEMP,1           ; get untagged wordcount
        lw      ARG2,total_allocation(ARG1)
        addu    ARG2,ARG2,ATEMP         ; inc the statistic.
        addui   ARG2,ARG2,1             ; (account for size fields too)
        sw      total_allocation(ARG1),ARG2

        or      ATEMP,ZERO,ARG3         ; move pointer to ATEMP

        lw      ARG3,-8(SP)             ; reload the temp registers
        lw      ARG2,-4(SP)             ;  & go zero block
        j       _ZBLOCK
        lw      ARG1,0(SP)
```

```
sov_msg:
        .ascii  "Stack overflow.  Dying..."
        .byte   0x0a,0x00
        .align 2
stack_overflow:
        lhi     ARG0,(printf_closure+1)>>16
        ori     ARG0,ARG0,(printf_closure+1)&0xffff
        lhi     ARG1,(sov_msg)>>16
        ori     ARG1,ARG1,(sov_msg)&0xffff
        lw      ATEMP, 3(ARG0)                  ; Jump to printf
        jalr    ATEMP                           ; /
        nop
        trap    0                               ; Die
        nop

        ;;  SFREE -- stack free
_SFREE:
        lw      ATEMP, -1(ATEMP)        ; Get (tagged) word count
        addu    SP,SP,ATEMP
        addu    SP,SP,ATEMP
        addui   SP,SP,4                 ; (Plus one for size slot).
        lhi     ATEMP,(TOTALMEMSIZE-4)>>16
        ori     ATEMP,ATEMP,(TOTALMEMSIZE-4)&0xffff
        sgt     ATEMP,SP,ATEMP
        bnez    ATEMP,stack_underflow
        nop
        jr      RETADR
        nop

suv_msg:
        .ascii  "Stack underflow.  Dying..."
        .byte   0x0a,0x00
        .align 2
stack_underflow:
        lhi     ARG0,(printf_closure+1)>>16
        ori     ARG0,ARG0,(printf_closure+1)&0xffff
        lhi     ARG1,(suv_msg)>>16
        ori     ARG1,ARG1,(suv_msg)&0xffff
        lw      ATEMP, 3(ARG0)                  ; Jump to printf
        jalr    ATEMP                           ; /
        nop
        trap    0                               ; Die
        nop

        ;;  _ALLOC -- allocate gc block on heap
        ;;     Same as SALLOC except on heap.  May initiate a garbage
        ;;    collection, which will change the value of pointers into
        ;;    the current space.
_ALLOC:
        sw      0(SP),ARG1              ; free-up a temp
        sw      -4(SP),ARG2             ; free-up another temp
        sw      -8(SP),ARG3             ; free-up yet another temp
```

```
        lhi     ARG1,(Memory_description)>>16
        ori     ARG1,ARG1,(Memory_description)&0xffff
        lw      ARG2,this_semispace_end(ARG1)   ; get end ptr ready...

        or      ARG3,ZERO,HP            ; get pointer to the new block
        addu    HP,HP,ATEMP             ; inc HP to next free mem
        addu    HP,HP,ATEMP             ; (ATEMP was tagged wordcount...)
        addu    HP,HP,4                 ; add a word for a size field

        sgt     ARG2,HP,ARG2            ; is hp over the end?
        beqz    ARG2,need_no_gc         ; if not, we're ok.
        nop

        ;; Uh-oh, we need to do a gc.
        lw      ARG3,-8(SP)             ; reload the temp registers
        lw      ARG2,-4(SP)             ;  & go collect (& retry alloc)
        j       gc
        lw      ARG1,0(SP)

need_no_gc:
        sw      -1(ARG3),ATEMP          ; stick the size into the new blk

        ;; Clear recursion-check flag to indicate it worked this time
        sw      just_did_a_gc(ARG1),ZERO

        ;; Increment statistics
        srli    ATEMP,ATEMP,1           ; change ATEMP to untagged wordcount
        lw      ARG2,total_allocation(ARG1)
        addu    ARG2,ARG2,ATEMP         ; inc the statistic.
        addui   ARG2,ARG2,1             ; (account for size fields too)
        sw      total_allocation(ARG1),ARG2
        lw      ARG2,total_allocs(ARG1)
        addui   ARG2,ARG2,1
        sw      total_allocs(ARG1),ARG2

        or      ATEMP,ZERO,ARG3         ; move pointer to ATEMP

        lw      ARG3,-8(SP)             ; reload the temp registers
        lw      ARG2,-4(SP)             ;  & go zero block
        j       _ZBLOCK
        lw      ARG1,0(SP)
```

```
;;  gc -- garbage collector
;;
;;  On entry:
;;    HP is garbage.
;;    ATEMP (still) contains the (tagged) number of words to allocate.
;;
;;  On return:
;;      All reachable data has been moved to the (old) other
;;    space, and the spaces have been swapped.  The root set
;;    is all registers except ZERO, HP, SP, ATEMP, and RETADR
;;      HP points to usable space in a (new) semispace
;;      ATEMP contains a pointer satisfying the alloc request
;;      All other regs are preserved modulo forwarding due to copying
;;      We return to address in RETADR

gc:
        sw      0(SP), ATEMP
        sw      -4(SP), RETADR
        subui   SP, SP, 8

        ;;  We'll push a frame onto the stack, making the root set be
        ;;  just FP and VAL (since VAL isn't saved into the frame).
        jal     _SALLOC                         ; Alocate frame on stack
        ori     ATEMP, ZERO, 2*(FrameSize)      ; (Tagged) size of frame
        sw      3(ATEMP),FP                     ; Link frame into
        or      FP,ZERO,ATEMP                   ;    dynamic chain
        jal     _SAVE
        sw      15(FP),ENV                      ; (Not saved by _SAVE)

        lhi     ARG1,(Memory_description)>>16
        ori     ARG1,ARG1,(Memory_description)&0xffff

        ;; Check to see if we recursed; if so, out of memory...
        lw      ARG2,just_did_a_gc(ARG1)
        bnez    ARG2,gc_loop_detected   ; EEEEEEEEEEEEEKKKKKKKKKKK!!!!!!
        nop

        ;; Flip the semispaces
        lw      ATEMP,this_semispace_end(ARG1)
        lw      HP,other_semispace_end(ARG1)
        sw      other_semispace_end(ARG1),ATEMP
        sw      this_semispace_end(ARG1),HP
        lw      ATEMP,this_semispace(ARG1)
        lw      HP,other_semispace(ARG1)
        sw      other_semispace(ARG1),ATEMP
        sw      this_semispace(ARG1),HP

        ;;  That cleverly left HP properly initialized.  Now scan root set.
        jal     maybe_copy
        or      ARG2,ZERO,FP
        or      FP,ZERO,ATEMP
```

```
jal     maybe_copy
or      ARG2,ZERO,VAL           ; VAL isn't in the frame...
or      VAL,ZERO,ATEMP

;; At this point, garbage collection is done.  Need to update
;; statistics, restore registers and the stack, and retry
;; allocation.

;; mark-up statistics...
lw      ARG2,num_gcs(ARG1)             ; inc gc count
addui   ARG2,ARG2,1
sw      num_gcs(ARG1),ARG2

lw      ARG3,this_semispace(ARG1)      ; get bottom of this space
subu    ARG3,HP,ARG3                   ; subtract it from current hp
srai    ARG3,ARG3,2                    ; change bytecount to wordcount
lw      ARG2,gc_words_copied(ARG1)     ; add this to total GC charge
addu    ARG2,ARG2,ARG3
sw      gc_words_copied(ARG1),ARG2

;; Set flag so next alloc (called from end of GC) will fail if no mem
sw      just_did_a_gc(ARG1),ARG1

;; Restore regs, pop activation frame from dyn chain, free mem it used
jal     _RESTORE
sw      15(FP),ENV                     ; (Not restored by _RESTORE)
or      ATEMP, ZERO, FP
lw      FP, 3(ATEMP)
jal     _SFREE
nop

;; Restore resgisters we saved right at start of gc
addui   SP, SP, 8
lw      RETADR, -4(SP)
lw      ATEMP, 0(SP)

;; retry the alloc, setting just_did flag so we can detect out-of-mem
j       _ALLOC                         ; go re-try the allocation...
nop
```

```
        ;; gc_loop_detected -- didn't free enough mem; print message and die
out_mem_msg:
        .ascii  "Insufficient memory to process alloc request.  Dying..."
        .byte   0x0a,0x00
        .align 2
gc_loop_detected:
        lhi     ARG0,(printf_closure+1)>>16
        ori     ARG0,ARG0,(printf_closure+1)&0xffff
        lhi     ARG1,(out_mem_msg)>>16
        ori     ARG1,ARG1,(out_mem_msg)&0xffff
        lw      ATEMP, 3(ARG0)                  ; Jump to printf
        jalr    ATEMP
        nop
        trap    0                               ; Die
        nop

        ;; maybe_copy
        ;; On entry:
        ;;   ARG1 points to Memory_description
        ;;   HP points to free memory in the (new) semispace (this_semispace)
        ;;   ARG2 contains a value to be copied (possibly)
        ;;
        ;; On exit:
        ;;     If thing in ARG2 is a tagged pointer, recursively maybe_copy
        ;;   each slot.  If that pointer points into the old semispace, copy
        ;;   object into new semispace.
        ;;     ATEMP is either old ARG2 if original wasn't a tagged
        ;;   pointer into old semispace, or pointer to new copied
        ;;   block if it was such a pointer.
maybe_copy:
        sw      0(SP),RETADR
        subui   SP,SP,4

        or      ATEMP,ZERO,ARG2         ; By default, return original value

        ;; If not a pointer, then we're done
        andi    ARG3,ARG2,1
        beqz    ARG3,maybe_copy_done
        nop

        ;; If pointer into this-space, just return it <== it's scanned already
        lw      ARG4,this_semispace(ARG1)
        sge     ARG3,ARG2,ARG4                  ; ARG3:= p >= this_semispace
        beqz    ARG3,check_4_semispace_ptr
        lw      ARG4,this_semispace_end(ARG1)
        slt     ARG3,ARG2,ARG4                  ; ARG3:= p < this_semispace_end
        bnez    ARG3,maybe_copy_done
        nop
```

```
        ;; pointer to otherspace => copy object AND scan object
        ;; pointer somewhere else => ONLY scan object
check_4_semispace_ptr:
        lw      ARG4,other_semispace(ARG1)
        sge     ARG3,ARG2,ARG4                  ; ARG3:= p >= other_semispace
        beqz    ARG3,scan_transitively
        lw      ARG4,other_semispace_end(ARG1)
        slt     ARG3,ARG2,ARG4                  ; ARG3:= p < other_semspace_end
        beqz    ARG3,scan_transitively
        nop

        ;; For semispace ptrs, don't chase if it's a forwarded pointer
is_semispace_ptr:
        lw      ATEMP,-1(ARG2)                  ; Get size or forwarding ptr
        andi    ARG4,ATEMP,1                    ; is this forwading pointer?
        bnez    ARG4,maybe_copy_done            ; yup ==> just return it.
        nop
live:   ;;  Ok, we've got a live one.
        or      ARG3,ZERO,ATEMP         ; get (tagged) size in words
        or      ATEMP,ZERO,HP           ; quick 'n dirty alloc
        addu    HP,HP,ARG3              ;   /
        addu    HP,HP,ARG3              ;  / (two times to get byte count)
        addui   HP,HP,4                 ; / (add 4 bytes for size slot)
        sw      -1(ATEMP),ARG3          ; stash block size (tagged in words)
        sw      -1(ARG2),ATEMP          ; set forwarding address.
        jal     copy_block              ; copy old block to new block
        nop

scan_transitively:
        ;;  Here, ATEMP points to a block whose contents need to be
        ;;  "maybe_copied".  loop over it, doing the proper thing.
        ;;  ARG2, ARG3 and ARG4 are free.
        subui   SP,SP,12                ; save 3 words live data thru recursion
        sw      4(SP), ATEMP            ; Save original value
        or      ARG3,ZERO,ATEMP         ; block pointer into ARG3
        lw      ARG4,-1(ARG3)           ; (tagged) wordcount in ARG4

scan_transitively_loop:
        beqz    ARG4,scan_transitively_done     ; quit when count is zero
        nop
        lw      ARG2,3(ARG3)            ; get a word
        sw      8(SP),ARG3              ; Save registers for recursive call
        sw      12(SP),ARG4             ;  /
        jal     maybe_copy              ; maybe_copy the item
        nop
        lw      ARG4,12(SP)             ; restore regs after recursive call
        lw      ARG3,8(SP)              ;  /
        sw      3(ARG3),ATEMP           ; replace old value
        addui   ARG3,ARG3,4             ; inc pointer
        j       scan_transitively_loop
        subui   ARG4,ARG4,2             ; decrement (tagged) count.
scan_transitively_done:
        lw      ATEMP, 4(SP)            ; Restore original value of pointer
        addui   SP,SP,12                ; Pop off spaced used during recursion
```

```
maybe_copy_done:
        addui   SP,SP,4
        lw      RETADR,0(SP)
        jr      RETADR
        nop

        ;; copy_block -- copy old block to new block.
        ;;   ARG2 points to old block (with forwarding ptr in size slot)
        ;;   ATEMP points to new block (with valid size in size block)
        ;;   ARG2, ARG3 and ARG4 are trashed; all rest (incl. ATEMP) presrved
copy_block:
        lw      ARG3,-1(ATEMP)          ; Get size in words (tagged)
        addu    ARG3,ARG3,ARG3          ;   * 2 =  untagged byte count
        addu    ARG2,ARG2,ARG3          ; point to end of old block
        addu    ARG3,ATEMP,ARG3         ; point to end of new block
copy_block_loop:
        subu    ARG4,ARG3,ATEMP         ; compare ptr to beginning of block
        beqz    ARG4,copy_block_done    ; done if no more to copy.
        nop
        lw      ARG4,-1(ARG2)           ; move a word
        sw      -1(ARG3),ARG4           ; /
        subui   ARG2,ARG2,4             ; adjust oldblock pointer
        j       copy_block_loop         ; (loop)
        subui   ARG3,ARG3,4             ; adjust newblock pointer
copy_block_done:
        jr      RETADR                  ; ARG2 and ATEMP are what they were.
        nop

        ;; statistics -- print memory statistics.
stat_format:
        .ascii  "#gc's=%d words copied by gc=%d words allocated=%d total allocs=%d"
        .byte   0x0a,0
        .align  2

STATISTICS:
        sw      0(SP), RETADR
        sw      4(SP), ARG0
        sw      8(SP), ARG1
        sw      12(SP), ARG2
        sw      16(SP), ARG3
        sw      20(SP), ARG4
        sw      24(SP), ARG5
        subui   SP,SP,28

        lhi     ARG0,(printf_closure+1)>>16
        ori     ARG0,ARG0,(printf_closure+1)&0xffff

        lhi     ARG1,(stat_format)>>16          ; Get format string
        ori     ARG1,ARG1,(stat_format)&0xffff  ; /
```

```
lhi     ATEMP,(Memory_description)>>16
ori     ATEMP,ATEMP,(Memory_description)&0xffff
lw      ARG2,num_gcs(ATEMP)              ; get count of GC's performed
slli    ARG2,ARG2,1                      ; tag the value for PRINTF
lw      ARG3,gc_words_copied(ATEMP)      ; get GC work estimate
slli    ARG3,ARG3,1                      ; tag the value for PRINTF
lw      ARG4,total_allocation(ATEMP)     ; get total amt allocated
slli    ARG4,ARG4,1                      ; tag the value for PRINTF
lw      ARG5,total_allocs(ATEMP)         ; get total calls to alloc
slli    ARG5,ARG5,1                      ; tag the value for PRINTF

;; Do call to printf
lw      ATEMP, 3(ARG0)                   ; Jump to printf
jalr    ATEMP
nop

addui   SP,SP,28
lw      ARG5,24(SP)
lw      ARG4,20(SP)
lw      ARG3,16(SP)
lw      ARG2,12(SP)
lw      ARG1,8(SP)
lw      ARG0,4(SP)
lw      RETADR,0(SP)
jr      RETADR
nop
```

D.3 runtime/epilog.s

The contents of the file runtime/epilog.s:

```
;; COMPILED CODE ENDS HERE .........................................

;;; epilog.s (in comp/backend/runtime)

;
;  Epilog:  What follows makes-up all run-time routines used by the microFX
;       system.  We let the DLX simulator do any necessary linking, and if
;       more code is included than is needed, so be it.  See macros.h
;       for register definitions.
;
;       prolog.s contains initialization code.
;

#include "macros.h"     /* registername macros, etc */
```

```
start1_closure: ;  Static closure for printf routine
        .word   4
        .word   START_1
        .word   0
printf_closure: ;  Static closure for printf routine
        .word   4
        .word   PRINTF
        .word   0
stats_closure: ;  Static closure for statistics printing routine
        .word   4
        .word   STATISTICS
        .word   0

#include "alloc.s"
#include "frames.s"
#include "printf.s"
#include "lib.s"

;  _endprogram
;       Does nothing, just marks the end of the program (and hence the
;       beginning of the heap at initialization time).  See alloc.s
;       to see how this is used.
;
_endprogram:
```

D.4 runtime/frames.s

The contents of the file runtime/frames.s:

```
;; frames.s
```

```
        ;;  _SAVE -- save registers into a frame
        ;;  C    try:
        ;;      FP contains a (tagged) pointer to the frame.
        ;;
        ;;  On exit:
        ;;      ATEMP and RETADR trashed
        ;;      r6--r29 are saved into frame according to convention
        ;;      All other regs are preserved
_SAVE:
        sw      111(FP),r29
        sw      107(FP),r28
        sw      103(FP),r27
        sw      99(FP),r26
        sw      95(FP),r25
        sw      91(FP),r24
        sw      87(FP),r23
        sw      83(FP),r22
        sw      79(FP),r21
        sw      75(FP),r20
        sw      71(FP),r19
        sw      67(FP),r18
        sw      63(FP),r17
        sw      59(FP),r16
        sw      55(FP),r15
        sw      51(FP),r14
        sw      47(FP),r13
        sw      43(FP),r12
        sw      39(FP),r11
        sw      35(FP),r10
        sw      31(FP),r9
        sw      27(FP),r8
        sw      23(FP),r7
        sw      19(FP),r6
        jr      RETADR
        nop
```

```
        ;;  _RESTORE -- restore regs from a frame.
        ;;  On entry:
        ;;      FP points to the frame
        ;;
        ;;  On exit:
        ;;      r6--r29 are restored from frame
        ;;      ATEMP and RETADR may be trashed
        ;;
_RESTORE:
        lw      r29,111(FP)
        lw      r28,107(FP)
        lw      r27,103(FP)
        lw      r26,99(FP)
        lw      r25,95(FP)
        lw      r24,91(FP)
        lw      r23,87(FP)
        lw      r22,83(FP)
        lw      r21,79(FP)
        lw      r20,75(FP)
        lw      r19,71(FP)
        lw      r18,67(FP)
        lw      r17,63(FP)
        lw      r16,59(FP)
        lw      r15,55(FP)
        lw      r14,51(FP)
        lw      r13,47(FP)
        lw      r12,43(FP)
        lw      r11,39(FP)
        lw      r10,35(FP)
        lw      r9,31(FP)
        lw      r8,27(FP)
        lw      r7,23(FP)
        lw      r6,19(FP)
        jr      RETADR
        nop
```

D.5 runtime/lib.s

The contents of the file runtime/lib.s:

```
;
;; micro-FX assembly-language libraries
;

        ;; Print (tagged) character in ARGO
        ;; callable from micro-FX [has type (-> (char) unit)]
PUTCHAR:
        lw      RETADR,11(FP)
        sub     VAL, VAL, VAL           ; Return value zero
        j       _TPUTCHAR
        nop
```

```
        ;;  _PUTCHAR -- put a character using dlx system call
        ;;  On entry:
        ;;    ATEMP is (untagged) character to print
        ;;  On exit:
        ;;    Stack used but SP preserved
        ;;    ATEMP trashed
putchar_format_string:
        .asciiz "%c"
        .align  2
putchar_format:
        .word   putchar_format_string
        .space  4
_TPUTCHAR: ;; Print (tagged) character in ATEMP.
        srai    ATEMP,ATEMP,1                   ; untag the char
_PUTCHAR: ;; Print (untagged) character in ATEMP.
        sw      0(SP), r14                      ; Save old value of r14
        subui   SP, SP, 4
        lhi     r14, (putchar_format>>16)       ; Pointer to args in r14
        ori     r14, r14, (putchar_format&0xffff)
        sw      4(r14), ATEMP
        trap    5                               ; Call print built-in
        nop
        addui   SP, SP, 4
        lw      r14, 0(SP)                      ; restore r14
        jr      RETADR                          ; Return to caller
        nop


        ;;  _SYM2STRING -- implements low-level part of sym->string
        ;;  On entry:
        ;;    ATEMP points to uFX symbol
        ;;
        ;;  On exit:
        ;;    Stack used but SP preserved
        ;;    ATEMP points to uFX string
_SYM2STRING:
        sw      0(SP),ARG0              ; free-up some temps
        sw      -4(SP),ARG1             ;   /
        sw      -8(SP),ARG2             ;  /
        sw      -12(SP),RETADR          ; /
        subui   SP, SP, 16

        or      ARG2, ATEMP, ZERO       ; Save pointer to symbol text
        subi    ARG1, ZERO, 1           ; Initialize length counter

len_loop:
        lbu     ARG0, 0(ATEMP)          ; Get next character in symbol
        addi    ARG1, ARG1, 1           ; Inc length counter
        addui   ATEMP, ATEMP, 1         ; Increment symbol pointer
        bnez    ARG0, len_loop          ; nxt char is not '\0', loop again
        nop
```

```
        slli    ATEMP, ARG1, 1          ; Tag length of string (D-slot!)
        jal     _ALLOC                  ; Allocate vector for string
        nop
        or      ARG1, ATEMP, ZERO       ; Save pointer returned by allocation

fill_loop:
        lbu     ARG0, 0(ARG2)           ; Get next character in symbol
        addui   ARG2, ARG2, 1           ; Increment pointer into symbol
        slli    ARG0, ARG0, 1           ; Tag the character
        beqz    ARG0, fill_done         ; If the character is '\0', quit loop
        nop
        sw      3(ARG1), ARG0           ; Store character in heap vector
        addui   ARG1, ARG1, 4           ; Increment pointer into heap vector
        j       fill_loop               ; repeat
        nop

fill_done:
        addui   SP, SP, 16
        lw      RETADR,-12(SP)          ; reload the temp registers
        lw      ARG2,-8(SP)             ;   /
        lw      ARG1,-4(SP)             ;  /
        lw      ARG0,0(SP)              ; /
        jr      RETADR                  ;  and return.
        nop
```

D.6 runtime/macros.h

The contents of the file runtime/macros.h:

```
/*
        macros.h: macros for easing the writing of prolog and epilog.
                do    /lib/cpp -P prolog.s > prolog.code
                      /lib/cpp -P epilog.s > epilog.code
                to use (done in the Makefile).
*/

#define ZERO    r0
#define VAL     r1
#define ENV     r2
#define FP      r3
#define SP      r4
#define HP      r5

#define ARG0    r6
#define ARG1    r7
#define ARG2    r8
#define ARG3    r9
#define ARG4    r10
#define ARG5    r11
#define ARG6    r12
#define ARG7    r13
#define ARG8    r14
```

```
#define ATEMP   r30
#define RETADR  r31

#define FrameSize       28              /* (untagged) # words in a frame. */

#define TOTALMEMSIZE    32768
```

D.7 runtime/printf.s

The contents of the file runtime/printf.s:

```
;; printf.s (in comp/backend/runtime)

        ;;  PRINTF -- A primative printf
        ;;  On entry:
        ;;    ARG1 points to a DLX string (uFX _symbol_) with grammar %x
        ;;  where x is:
        ;;      x ::=  %       a '%' sign
        ;;             d       (decimal) integer
        ;;             b       boolean
        ;;             c       char
        ;;             s       string (null-terminated)
        ;;             F       function
        ;;             pxx     pairs (where x is recursive format)
        ;;             lx      lists (thus a list of lists of ints is
        ;;                            printed by %lld)
        ;;             rx      refs
        ;;             vx      vectors
        ;;    ARG2--ARG8 are rest of printf arguments (up to 6)
        ;;
        ;;  On exit:
        ;;    Uses regular uFX calling conventions, ie, almost nothing saved.
        ;;    Returns the-unit.
PRINTF:
        nop
printf_loop:
        lb      VAL,0(ARG1)
        beqz    VAL,end_printf
        nop
        seqi    r29,VAL,0x25    ; '%'
        beqz    r29,no_escape
        nop

        ;;  got a %, see what the esc'd char is...
        addui   ARG1,ARG1,1
        lbu     VAL,0(ARG1)
        beqz    VAL,end_printf
        nop
```

```
        seqi    r29,VAL,0x62    ; 'b'
        bnez    r29,bool_out
        nop

        seqi    r29,VAL,0x63    ; 'c'
        bnez    r29,char_out
        nop

        seqi    r29,VAL,0x64    ; 'd'
        bnez    r29,dec_out
        nop

        seqi    r29,VAL,0x70    ; 'p'
        bnez    r29,pair_out
        nop

        seqi    r29,VAL,0x6c    ; 'l'
        bnez    r29,list_out
        nop

        seqi    r29,VAL,0x76    ; 'v'
        bnez    r29,vec_out
        nop

        seqi    r29,VAL,0x72    ; 'r'
        bnez    r29,ref_out
        nop

        seqi    r29,VAL,0x73    ; 's'
        bnez    r29,sym_out
        nop

        ;; Otherwise, just print the escaped character
no_escape:
        jal     _PUTCHAR                ; Print the character
        or      ATEMP, VAL, ZERO
        j       printf_loop
        addui   ARG1,ARG1,1

end_printf:
        lw      RETADR, 11(FP)
        jr      RETADR
        or      VAL, ZERO, ZERO

shiftem:
        or      ARG2,ZERO,ARG3
        or      ARG3,ZERO,ARG4
        or      ARG4,ZERO,ARG5
        or      ARG5,ZERO,ARG6
        or      ARG6,ZERO,ARG7
        jr      RETADR
        or      ARG7,ZERO,ARG8
```

```
        ;; To print a char, untag it and use _PUTCHAR
char_out:
        jal     shiftem         ; Shift arguments down
        srai    ATEMP,ARG2,1    ; Untag char to be printed (in D-slot!)
        jal     _PUTCHAR
        nop
        j       printf_loop
        addui   ARG1, ARG1, 1


        ;; Use DLX's printf with %d format
dec_out:
        lhi     r14,(dec_format)>>16
        ori     r14,r14,(dec_format)&0xffff     ; Load-up the %d format string
        srai    VAL,ARG2,1                      ; Untag int 2B printed
        sw      4(r14),VAL
        trap    5
        nop
        jal     shiftem                         ; shift the args around...
        nop
        j       printf_loop
        addui   ARG1,ARG1,1
dec_format_string:
        .asciiz "%d"
        .align  2
dec_format:
        .word   dec_format_string
        .space  4                       ; Save space for number to print


        ;; Use _PUTCHAR to print a # then a T or F
bool_out:
        jal     _PUTCHAR                ; Print '#'
        ori     ATEMP, ZERO, 0x23       ; /
        sne     VAL, ARG2, ZERO
        ori     ATEMP, ZERO, 14
        sll     ATEMP, ATEMP, VAL
        addi    ATEMP, ATEMP, 56        ; ATEMP = (VAL ? 'T' : 'F')
        jal     _PUTCHAR
        nop
        jal     shiftem
        nop
        j       printf_loop
        addui   ARG1,ARG1,1
```

```
        ;; Use DLX printf here.
sym_out:
        lhi     r14,(string_format)>>16
        ori     r14,r14,(string_format)&0xffff
        sw      4(r14),ARG2
        trap    5
        nop
        jal     shiftem                 ; shift the args around...
        nop
        j       printf_loop
        addui   ARG1,ARG1,1
string_format_string:
        .asciiz "%s"
        .align  2
string_format:
        .word   string_format_string
        .space  4
```

```
        ;;  build_format -- build format for recursive call to printf
        ;;  On entry:
        ;;      ARG1 points to first character of the old format
        ;;   (excluding % sign).  This char is assumed to be one of
        ;;   the compound types (l, v, p, r).
        ;;  On exit:
        ;;      ATEMP points to new format (allocated on stack).
        ;;      ARG1 points to last character of old format
        ;;      r14,r15,r16,r17,r18 trashed
build_format:
        ; first calculate length of new format string (minus the % char)
        or      r14, ARG1, ZERO
        ori     ATEMP, ZERO, 0
        ori     r18, ZERO, 0
bf_deeper:
        addi    r18, r18, 1             ; Counts nested p's
bf_len_loop:
        addui   r14, r14, 1
        lbu     r15, 0(r14)
        addi    ATEMP, ATEMP, 1
        sequi   r16, r15, 0x70          ; 'p'
        bnez    r16, bf_deeper
        sequi   r16, r15, 0x6c          ; 'l'
        bnez    r16, bf_len_loop
        sequi   r17, r15, 0x72          ; 'r'
        bnez    r17, bf_len_loop
        sequi   r16, r15, 0x76          ; 'v'
        bnez    r16, bf_len_loop
        nop
        subi    r18, r18, 1
        bnez    r18, bf_len_loop
        nop
bf_len_done:    ; length in ATEMP
        ; Now allocate room on stack for the new format string and the length
        ; of the string
        addi    r14, ATEMP, 8           ; Include room for '%' & a length
        andi    r14, r14, 0xfffc        ; Align length to word boundary
        subu    SP, SP, r14
        sw      4(SP), r14              ; Save length to make popping easy
        ; Create new format string by appending a '%' to thing passed in
        addui   r14, SP, 8              ; Ptr to base of new string
        ori     r15, ZERO, 0x25
        sb      0(r14), r15             ; Store a '%'
bf_cp_loop:
        addui   ARG1, ARG1, 1
        lbu     r15, 0(ARG1)
        addui   r14, r14, 1
        subi    ATEMP, ATEMP, 1
        sb      0(r14), r15
        bnez    ATEMP, bf_cp_loop
        nop
bf_done:
        addui   r14, r14, 1             ; Terminate format string
        sb      0(r14), ZERO
        jr      RETADR
        addui   ATEMP, SP, 8            ; Return pointer to new format string
```

```
_pf_recurse:
        j       PRINTF
        sw      11(FP), RETADR


        ;; Call PRINTF recursively to print the thing pointed to
pair_out:
        lhi     r14,(pair_hdr)>>16       ; Print "(pair "
        ori     r14,r14,(pair_hdr)&0xffff
        trap    5
        nop

        ;; Set up stack-frame for recursive call
        jal     _SALLOC
        ori     ATEMP, ZERO, (FrameSize*2)
        sw      3(ATEMP), FP                     ; Old static chain
        or      FP,ZERO,ATEMP
        addi    ATEMP, SP, (FrameSize+1)*4       ; Old SP
        sw      7(FP), ATEMP                     ; /

        ;; Print thing on left
        jal     build_format             ; Returns ptr to new string in ATEMP
        nop                              ; & ARG1 pting 2 1st chr in old format
        or      r29, ZERO, ATEMP         ; Stash this away for a second
        jal     _SAVE                    ; Save reg's during recursive call
        nop
        or      ARG1, ZERO, r29          ; Save new format string
        lw      ARG2,3(ARG2)             ; Pass item to print in ARG2
        jal     _pf_recurse              ; Recursive call to printf
        nop
        jal     _PUTCHAR                 ; Put space between items
        ori     ATEMP, ZERO, 0x20
        jal     _RESTORE                 ; Get back our old registers
        nop
        ;; Now do right hand thing
        jal     build_format             ; Returns ptr to new string in ATEMP
        nop                              ; & ARG1 pting 2 1st chr in old format
        or      r29, ZERO, ATEMP         ; Stash this away for a second
        jal     _SAVE                    ; Save reg's during recursive call
        nop
        or      ARG1, ZERO,r29           ; Save new format string
        lw      ARG2,7(ARG2)             ; Pass item to print in ARG2
        jal     _pf_recurse              ; Call PRINTF recursively
        nop
        jal     _PUTCHAR                 ; Put closing ')'
        ori     ATEMP, ZERO, 0x29
        jal     _RESTORE
        nop
```

```
        ;; Finish up
        lw      SP,7(FP)                ; Get rid of activation frame.
        lw      FP,3(FP)                ; /
        jal     shiftem                 ; Shift args down
        nop
        j       printf_loop
        addui   ARG1, ARG1, 1
pair_hdr_string:
        .asciiz "(pair "
        .align  2
pair_hdr:
        .word   pair_hdr_string
        .space  4


        ;; Call PRINTF recursively to print the list items.
list_out:
        jal     _PUTCHAR                ; Put a '('
        ori     ATEMP, ZERO, 0x28

        ;; Set up stack-frame for recursive call
        jal     _SALLOC
        ori     ATEMP, ZERO, (FrameSize*2)
        sw      3(ATEMP), FP                    ; Old static chain
        or      FP,ZERO,ATEMP
        addi    ATEMP, SP, (FrameSize+1)*4      ; Old SP
        sw      7(FP), ATEMP                    ; /

        jal     build_format            ; Returns ptr to new string in ATEMP
        nop                             ; & ARG1 pting 2 1st chr in old format
        or      r29,ZERO,ATEMP          ; Save ptr to new string in stack slot

        beqz    ARG2,list_done          ; If list is empty, skip loop
        nop
        j       list_nospace            ; skip space in front of first item
        nop
```

```
list_loop:
        jal     _PUTCHAR                ; Print a space in front of item
        ori     ATEMP, ZERO, 0x20       ;  (in D-slot!)
list_nospace:
        jal     _SAVE
        nop
        or      ARG1,ZERO,r29           ; Pass format string in ARG1
        lw      ARG2,3(ARG2)            ; Pass item to print in ARG2
        jal     _pf_recurse             ; Call PRINTF recursively
        nop
        jal     _RESTORE
        nop
        lw      ARG2,7(ARG2)
        bnez    ARG2,list_loop          ; do again if more items in list
        nop
list_done:
        lw      SP,7(FP)                ; Get rid of activation frame.
        lw      FP,3(FP)                ; /
        jal     _PUTCHAR                ; Print closing ')'
        ori     ATEMP, ZERO, 0x29
        jal     shiftem                 ; Shift args down
        nop
        j       printf_loop
        addui   ARG1, ARG1, 1


        ;; Call PRINTF recursively to print the vec items.
vec_out:
        jal     _PUTCHAR                ; Put a '#'
        ori     ATEMP, ZERO, 0x23
        jal     _PUTCHAR                ; Put a '('
        ori     ATEMP, ZERO, 0x28

        ;; Set up stack-frame for recursive call
        jal     _SALLOC
        ori     ATEMP, ZERO, (FrameSize*2)
        sw      3(ATEMP), FP                    ; Old static chain
        or      FP,ZERO,ATEMP
        addi    ATEMP, SP, (FrameSize+1)*4      ; Old SP
        sw      7(FP), ATEMP                    ; /

        jal     build_format            ; Returns ptr to new string in ATEMP
        nop                             ; & ARG1 pting 2 1st chr in old format
        or      r28,ZERO,ATEMP          ; Save new format string

        lw      r29, -1(ARG2)           ; Get length of vector
        beqz    r29, vec_done           ; If vec is empty, skip loop
        nop
        j       vec_nospace             ; skip space in front of first item
        nop
```

```
vec_loop:
        jal     _PUTCHAR                ; Print a space in front of item
        ori     ATEMP, ZERO, 0x20       ;  (in D-slot!)
vec_nospace:
        jal     _SAVE
        nop
        or      ARG1,ZERO,r28           ; Pass format string in ARG1
        lw      ARG2,3(ARG2)            ; Pass item to print in ARG2
        jal     _pf_recurse             ; Call PRINTF recursively
        nop
vec_return:
        jal     _RESTORE
        nop
        addui   ARG2,ARG2,4             ; Advance to next item in vec
        subi    r29,r29,2               ; do a _tagged_ subtract of 1
        bnez    r29,vec_loop            ; do again if more items in vec
        nop
vec_done:
        lw      SP,7(FP)                ; Get rid of activation frame.
        lw      FP,3(FP)                ; /
        jal     _PUTCHAR                ; Print closing ')'
        ori     ATEMP, ZERO, 0x29
        jal     shiftem                 ; Shift args down
        nop
        j       printf_loop
        addui   ARG1, ARG1, 1



        ;; Call PRINTF recursively to print the thing pointed to
ref_out:
        lhi     r14,(ref_hdr)>>16       ; Print "(ref "
        ori     r14,r14,(ref_hdr)&0xffff
        trap    5
        nop

        ;; Set up stack-frame for recursive call
        jal     _SALLOC
        ori     ATEMP, ZERO, (FrameSize*2)
        or      FP,ZERO,ATEMP
        sw      3(ATEMP), FP                    ; Old static chain
        addi    ATEMP, SP, (FrameSize+1)*4      ; Old SP
        sw      7(FP), ATEMP                    ; /

        jal     build_format            ; Returns ptr to new string in ATEMP
        nop                             ; & ARG1 pting 2 1st chr in old format
        or      r29, ATEMP, ZERO        ; Save new format string

        jal     _SAVE
        or      ARG1,ZERO,r29
        lw      ARG2,3(ARG2)            ; Pass item to print in ARG2
        jal     _pf_recurse             ; Call PRINTF recursively
        nop
```

```
        lw      SP,7(FP)                ; Get rid of activation frame.
        lw      FP,3(FP)                ; /
        jal     _RESTORE
        nop
        jal     _PUTCHAR                ; Put closing ')'
        ori     ATEMP, ZERO, 0x29
        jal     shiftem                 ; Shift args down
        nop
        j       printf_loop
        addui   ARG1, ARG1, 1
ref_hdr_string:
        .asciiz "(ref "
        .align  2
ref_hdr:
        .word   ref_hdr_string
        .align  2
```

D.8 runtime/prolog.s

The contents of the file runtime/prolog.s:

```
;;; prolog.s (in comp/backend/runtime)
;
; prolog.s: microFX program prolog.  This gets prefixed to every compiled file.
;

#include "macros.h"     /* registername macros, etc */

;;  Execution starts right here!
        .global PRINTF
        .global STATISTICS
        ;; ^ is needed due to dlxsim bug :-(
```

```
program_1:
        ;; zero regs so that gc stays happy (all reg's have tagged values)
        or      r1,ZERO,ZERO
        or      r2,ZERO,ZERO
        or      r3,ZERO,ZERO
        or      r4,ZERO,ZERO
        or      r5,ZERO,ZERO
        or      r6,ZERO,ZERO
        or      r7,ZERO,ZERO
        or      r8,ZERO,ZERO
        or      r9,ZERO,ZERO
        or      r10,ZERO,ZERO
        or      r11,ZERO,ZERO
        or      r12,ZERO,ZERO
        or      r13,ZERO,ZERO
        or      r14,ZERO,ZERO
        or      r15,ZERO,ZERO
        or      r16,ZERO,ZERO
        or      r17,ZERO,ZERO
        or      r18,ZERO,ZERO
        or      r19,ZERO,ZERO
        or      r20,ZERO,ZERO
        or      r21,ZERO,ZERO
        or      r22,ZERO,ZERO
        or      r23,ZERO,ZERO
        or      r24,ZERO,ZERO
        or      r25,ZERO,ZERO
        or      r26,ZERO,ZERO
        or      r27,ZERO,ZERO
        or      r28,ZERO,ZERO
        or      r29,ZERO,ZERO
        or      r30,ZERO,ZERO

        jal     _init_runtime                   ; Init runtime sys
        nop

        ;; Set-up an initial frame to return through
        ori     ATEMP, ZERO, FrameSize
        jal     _ALLOC
        nop
        or      FP, ATEMP, ZERO                 ; FP points to frame
        sw      3(FP),ZERO
        sw      7(FP),SP
        lhi     ATEMP,(print_res)>>16           ; place to return to
        ori     ATEMP,ATEMP,(print_res)&0xffff  ;  /
        sw      11(FP),ATEMP                    ; /
        sw      15(FP),ZERO
        jal     _SAVE
        nop
```

```
        ;; jump to START_1 through statically-created closure.
        lhi     ARG0,(start1_closure+1)>>16     ; Get closure to printf
        ori     ARG0,ARG0,(start1_closure+1)&0xffff ; /
        lw      ATEMP, 3(ARG0)                  ; Jump to START_1
        jr      ATEMP
        nop

print_res:
        ;; Print result by calling printf. Recall that the compiler
        ;; gave us a string "RESULT_FORMAT" which is a printf format
        ;; for the type of the result.  (Reuses activation from from
        ;; above.)
        lhi     ATEMP,(print_stat)>>16          ; place for printf to return to
        ori     ATEMP,ATEMP,(print_stat)&0xffff ;  /
        sw      11(FP),ATEMP                    ; /
        lhi     ARG0,(printf_closure+1)>>16     ; Get closure to printf
        ori     ARG0,ARG0,(printf_closure+1)&0xffff ; /
        lhi     ARG1,(RESULT_FORMAT)>>16        ; put the format string in ARG1
        ori     ARG1,ARG1,(RESULT_FORMAT)&0xffff ; /
        or      ARG2,ZERO,VAL                   ; put the returned val in ARG2
        lw      ATEMP, 3(ARG0)                  ; Jump to printf
        jr      ATEMP                           ; /
        nop

print_stat:
        ;; Print a couple of new-lines
        ori     ATEMP, ZERO, 10
        jal     _PUTCHAR
        nop
        ori     ATEMP, ZERO, 10
        jal     _PUTCHAR
        nop

        ;; Call STATISTICS to print stat's
        lhi     ATEMP,(_EXIT)>>16               ; place for stats to return to
        ori     ATEMP,ATEMP,(_EXIT)&0xffff      ;  /
        sw      11(FP),ATEMP                    ; /
        lhi     ARG0,(stats_closure+1)>>16      ; Get closure to stats printing
        ori     ARG0,ARG0,(stats_closure+1)&0xffff
        lw      ATEMP, 3(ARG0)                  ; Jump to statistics
        jr      ATEMP                           ; /
        nop

_EXIT:
        trap    0                               ; "exit" sys call
        nop

;; COMPILED CODE STARTS HERE........................................
```

References

[1] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. Report on the FX-91 Programming Language. 1991. To be published in SIGPLAN Notices.

[2] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture — A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

Index